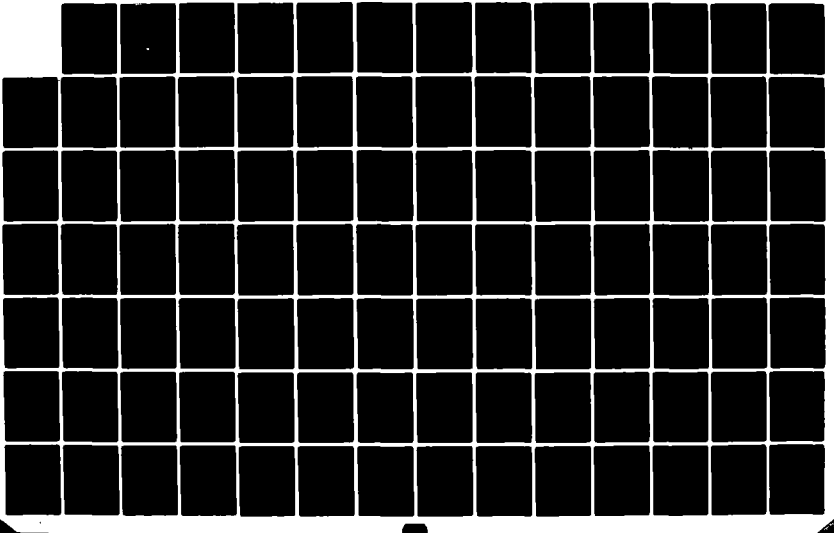
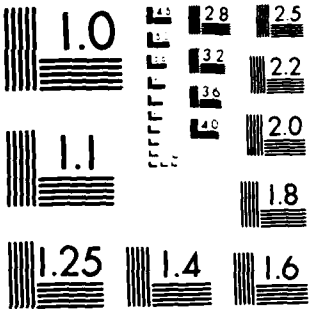
AD-A100 189 ON THE FORMAL SPECIFICATION OF COMPUTER COMMUNICATION PROTOCOLS(U) MARYLAND UNIV COLLEGE PARK DEPT OF COMPUTER SCIENCE K F SHOTTING DEC 80 TR-973 1/2

UNCLASSIFIED AFOSR-TR-81-0491 AFOSR-78-3654 F/G 9/2 NL





LEVEL II

12

COMPUTER SCIENCE TECHNICAL REPORT SERIES

UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742

LEVEL II

12

Technical Report TR-973 December 1980
AFOSR-78-3654

On The Formal Specification Of Computer Communication Protocols

Kenneth F. Shotting

Abstract

This thesis gives a framework for specifying and verifying communications protocols. It views a communication system as a hierarchy of abstract machines using the Hierarchical Development Methodology of SRI International. The major results are a framework for formally specifying protocols, the formal specification of a particular protocol and an evaluation of the applicability of HDM to protocol specifications. It includes a specification of the Internet Protocol developed for the Department of Defense as a sample use of the methods advocated in the thesis.

This research was supported in part by the Air Force Office of Scientific Research under grant AFOSR-78-3654.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: AFOSR-TR-81-0491	2. GOVT ACCESSION NO.: AD-A100 189
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): ON THE FORMAL SPECIFICATION OF COMPUTER COMMUNICATION PROTOCOLS.	5. TYPE OF REPORT & PERIOD COVERED: INTERIM
	6. PERFORMING ORG. REPORT NUMBER: TR-973
7. AUTHOR(s): Kenneth F. Shotting	8. CONTRACT OR GRANT NUMBER(s): AFOSR-78-3654
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Department of Computer Science, University of Maryland, College Park MD 20740	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: PE61102F, 2304/A2
11. CONTROLLING OFFICE NAME AND ADDRESS: Air Force Office of Scientific Research/NM, Bolling AFB DC 20332	12. REPORT DATE: DECEMBER 1980
	13. NUMBER OF PAGES: 95
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This thesis gives a framework for specifying and verifying communications protocols. It views a communication system as a hierarchy of abstract machines using the Hierarchical Development Methodology of SRI International. The major results are a framework for formally specifying protocols, the formal specification of a particular protocol and an evaluation of the applicability of HDM to protocol specifications. It includes a specification of the Internet Protocol developed for the Department of Defense as a sample use of the methods advocated in the thesis.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

401 044

Table of Contents

Figures

ON THE FORMAL SPECIFICATION OF COMPUTER COMMUNICATION PROTOCOLS

by

Kenneth F. Shotting

1. INTRODUCTION

Computer networking, though only slightly over ten years old, has already become widespread. Numerous networks now exist using various networking technologies and communication protocols. It has given rise to a whole new specialized area in the realm of computer engineering and science. Companies have been formed over this period, such as TELENET and TYMSHARE, with their major purpose being to offer networking services to clients. Interest is growing in having the ability to interconnect networks using different technologies.

The goal of internetworking is to allow a user on any system on any network to access data or services on any other system in as transparent a manner as possible. This is a long term goal, however a model for such a system exists. The term catenet (concatenated network) has been coined for a collection of connected networks [Cer78].

The interconnection of networks leads to many technical problems. Although different networks may provide similar services, it is often

difficult to map these similar services across networks because of differences in the underlying protocols and the lack of a good definition of the protocols. Quite often the best definition of the protocols is the running code. Problems of this nature point out the need for an understandable definition of communications protocols.

Network interconnection is not completely responsible for such problems, it just exacerbates them; even without network interconnection problems exist. Problems are known to exist on the ARPAnet (1).

Although some problems can be attributed to "growing pains", many are just another manifestation of the general probelms experienced by software -- protocols must ultimately be cast into software. As such protocol implementation efforts have suffered the normal problems suffered by large software systems: the software tends to be complex, expensive, and difficult to understand, modify or maintain and late.

---

(1) A few examples. The ARPAnet Network Control Protocol (NCP) has occasionally dropped or duplicated parts of messages. The ARPAnet File Transfer Protocol (FTP) has truncated and garbled files being transferred. The FTP problems rarely occur when using two systems running essentially the same code (although problems doubtlessly exist in this case), but between two systems running different implementations of the FTP protocol (e.g. one for TOPS-20; one for UNIX). In one case getting a file from the TOPS-20 system then sending it back resulted in differences, whereas the same setting used to send the file to the UNIX system then get it back gave an identity mapping. In another instance the FTP protocol was randomly setting "unused" bits. Admittedly the transfer was somewhat nonstandard. The file was an executable file stored as ASCII characters; however, neither the TOPS-20 nor the UNIX documentation warned that problems could occur if a character file transfer used the full ASCII character set. Also, these were not occasional network errors, but repeatable occurrences.

Our goal in this thesis is to explore the use of formal development techniques in the process of designing and implementing protocols. Since it is easy to overlook difficulties or gloss over significant problems with plausible but specious explanations by working with toy examples, we will examine existing methodologies and protocols. It is not meant to be either a protocol or methodology development effort but an evaluation. The intent is to analyze what currently exists and to make an effort to determine what changes and improvements must be made in each area to have an effective melding of the two technologies. This also attempts to approach the problem realistically from another point of view. Actual problems rarely allow the freedom to start from scratch. Normal practice involves making things work together in the existing environment due to cost, scheduling and/or existing capital investment. The difference between designing and verifying a system with complete freedom versus with external constraints is very great.

We will investigate techniques that have been developed to specify and verify the properties of concurrent programs. We will discuss current needs in the area of protocol specification and verification starting with what relationships exist between techniques for verifying communication protocols and those necessary to verify other large software projects such as operating systems. (Since verification is the process of demonstrating a system consistent with its specification henceforth we will assume that verification implies specification to avoid constantly using the phrase "specify and verify".)

Although the verification of large software systems is not currently feasible, interest in verifying such systems continues to grow. There are three types of software whose verification is recognized to be of major importance: operating systems, communication protocols, and data base management systems. In a system built along clear levels of abstraction these three are responsible for three types of global information management. The communication software is responsible for information transfer, the operating system is responsible for resource management and the data base software is responsible for information management. The reason for the importance of these particular types of software is that they provide global interactions among users and programs. Their incorrect operation can affect many users whereas the incorrect operation of an application program is normally localized. There is still interest in verifying such application programs, in fact more verification has been done on them because they tend to be smaller and easier to specify. In this thesis, however, we are interested in looking at how this technology can be applied to communication systems and where improvements need to be made to make it practical, not just for simple examples, but for something as complex as a communication system.

2. SPECIFICATION AND VERIFICATION

From a verification perspective a system has the structure shown in figure 1. Between each of the boxes there is a verification step to show that the next level properly maps the higher level into the

next lower level. The first correspondence step verifies that the software specifications correspond to the given formal model of the requirements. In the second correspondence step the executable program is verified to correctly implement the specifications. If these steps are successfully carried out, the software can be said to correctly implement the modeled system.

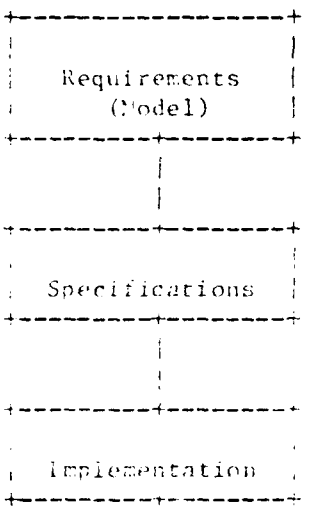


Fig. 1

The box representing the specifications may be (and for large systems must be) further decomposed into levels. The requirements for a particular level will be derived from the specifications of the level above it; its implementation will be in terms of the interface provided by the level below it. The interface at a given level provides a set of functions different from the functions provided at the base hardware. These functions characterize the behavior in terms of the concepts of the level being defined. Such an interface is called an abstract machine and the interface represents a level of

abstraction (i. e. a view of the system in terms of a set of functions which are easier to grasp than the detailed implementation).

Formal specifications are often viewed as serving two purposes. First, as providing a complete and rigorous description of the system being designed and second, as providing a basis for mathematically proving that the software implementation meets the design. For the latter use the precise mathematical formalism is admirably suited; for the former the necessary formal can obscure the intended functionality and purpose.

Axiomatic specifications can lead to good concise models. Axiomatic approaches are normally used in mathematical systems because they allow a theory to be developed logically, step-by-step from a set of assumptions which form the basis for the theory.

Axioms tend to define a function in terms of its overall properties; sometimes such definitions are not intuitive. As an example consider the following three axioms:

```
n>0 -> f(n,n)=n
f(n,m)=f(m,n)
f(n,m)=f(n+m,m))
```

These axioms are necessary and sufficient to define the Greatest Common Divisor (GCD) function. They are well known in the realm of program verification; however, someone not familiar with these axioms may spend a significant amount of time trying to determine the function (if any) which they define.

An abstract machine gives a function defined in terms of its inputs and outputs. An abstract machine can be used in a specification as an intermediate step between axioms and code. The following abstract machine definition of the GCD function gives a clearer idea of the function to be implemented without actually giving an implementation of the function.

```
f(INTEGER i, j) -> INTEGER k;
k = MAX({x | (i MOD x = 0) AND (j MOD x = 0)});
```

An operational specification is essentially a definition by example. It is very constraining on the implementor as it gives no information as to whether or not a equivalent algorithm for the same function is acceptable. The following is an operation specification for the GCD function; however, many other algorithms implement the same function.

```
f(i, j){
while i > 0 do
        if j >= i then j = j-i else swap(i, j);
result = j;}
```

The techniques discussed in the preceeding paragraghs range from the very abstract to the very concrete. They are not orthogonal but hierarchical. When a subject is not well understood the only possible specification may be an operational one. As the area evolves and progresses, it is possible to migrate towards more abstract specifications until necessary and sufficient conditions for the subject area have been established.

The difference between these types of specifications is mostly one of philosophy. The specification techniques, though intended for a

specific approach, are usually general enough to allow all three forms of specifications to be written using its language. Often, though, the verification technique and tools are such that attempting to verify a program using other than the intended approach becomes very tedious, if possible at all.

a. Choosing a Methodology

Our goal is to examine the use of formal methods in developing protocols. The first step towards this goal is to choose the overall methodology we will use for design, implementation and verification. Some work has been done in an area generally called protocol verification. A few survey papers have been written describing the techniques which are available [Sun78, TAL79]. The techniques described range from those intended to examine very protocol specific features to those intended for developing general-purpose software.

As the overall methodology will guide the whole process it should be chosen with care using well thought out criteria. The following criteria seem essential for the goals we wish to accomplish. First, communication protocols are normally developed in layers. A methodology which provides a good layering structure would be beneficial to the effort. Second, the methodology should be applicable to types of software other than protocols. Protocols are not developed in isolation; it should be possible to integrate protocol specifications into other formal specification efforts, especially in the areas of operating systems and data base management systems.

Protocol Layer	Functions
6. Application	Funds transfer, Information retrieval, Electronic mail, ...
5. Utility	File Transfer, Virtual terminal support, ...
4. End/End Subscriber	Interprocess communication (e. g. Virtual circuit, Datagram, Real time, Broadcast, ...)
3. Network Access	Network access services (e. g. Virtual circuit, Datagram, ...)
2. Intranet End/End	Flow control, Sequencing
1. Intranet Node/Node	Congestion control, routing
0. Link Control	Error handling, Link flow control

Fig. 2

Figure 2 [CK78] is an example of protocol layering. In [CK78] the authors analyze current network architectures and describe the ways in which a selection of current networks fit into this framework. This type of analysis is the major reason we believe any attempt to verify protocols should clearly take this structure into account. In some sense this should not prove to be a problem as the layering of protocols is in response to current software design philosophies which advocate decomposing programs into manageable units. Although each layer is still a very complex unit, the decomposition shown in figure 2 is a reasonable initial decomposition. In particular, each layer can be viewed as an abstract machine on which the layer above it executes. These layers can in turn be further decomposed into a set of

simpler abstract machines.

Designing software as a hierarchy of abstract machines is a well-known technique in the realm of operating systems. This technique for designing operating systems was pioneered by Dijkstra in the THE operating system [Dij68] and has been used in many projects since. For the same reasons that this approach is advocated as good practice for the design of understandable, reliable software, abstract machine architectures are used in some verification approaches. The abstract machine approach advocates decomposing a large monolithic software program into a set of programs each of which is less complex and more understandable than the original. In decomposing the design into many small, understandable pieces, it also decomposes the theorems which are generated into many small, easier to verify theorems.

Over the last several years significant effort has been put into designing and implementing trusted (secure) operating systems. With the current state of computer networking, such operating systems will need to have the ability to manage network resources. One of the major techniques being used, in attempts to develop trusted operating system, is formal specification. Such operating systems will require network software which is formally specified. After integrating the network software the system should retain the level of trust (security) it previously had. This would require the ability to integrate the protocol and operating system formal specifications.

b. Specification and verification methodologies

We will consider four methodologies in this section. The criteria for selecting these four for discussion were that they are in the public domain, that reasonable documentation of each methodology exists and that the tools associated with each methodology have been developed to the point that they are useable by someone not intimately involved with the development effort. The methodologies we will consider are the Gypsy methodology of the University of Texas at Austin, the Hierarchical Development Methodology of SRI International, the Affirm system from USC-ISI, and the Stanford University Pascal verifier. A survey which includes a description of these methodologies is available [CGHM80]. As part of this survey all of the methodologies were used to specify some very simple top level properties of a security kernel.

1. Gypsy methodology

The Gypsy methodology has been under development at the University of Texas by the Certifiable Minicomputer Project (CMP) since 1974. The specifications are based on the axiomatic approach. Development proceeds from a top-level specification through successive refinements.

The Gypsy language is for both specification and implementation. It is well suited for protocols in two ways. Its model of interprocess communication is based on message buffers and particular attention has been paid to concurrency.

The Gypsy Verification Environment (GVE) is a well integrated system to aid the user in the design and verification process. It has an excellent user interface. The major weakness of the GVE is the theorem prover. As part of the proof process the system generates many fairly trivial theorems. Driving these proofs through the system is rather time consuming and tedious. Theorem proving problems exist to one degree or another in all the systems.

Developing a system using Gypsy can proceed in many ways, but the advocated way is a top-down, stepwise refinement approach. GVE provides an incremental development and verification capability which allows some one using the recommended procedure to write the top level, write the interfaces to the next lower level, and then prove the correctness of the top level. This procedure can be applied recursively at each level to assure the correctness of the system at each step of the development.

Gypsy has been used to model a small subset of the ARPAnet [Wel76] and work is being started under CMP to investigate the use of Gypsy for protocol verification [Div80]. Gypsy has also been used to develop a small process for examining and approving for downgrading portions of classified documents which reside on computer systems. Texas Instruments has used Gypsy as part of an exploratory effort for NASA to develop very reliable software for an air traffic control system.

2. Hierarchical Development Methodology (HDM)

HDM is a methodology developed at SRI International for the design of large hardware/software systems [Rob79, LRS79]. The

methodology begins with a top-level description of the system's requirements. The system is decomposed into modules and the modules are organized hierarchically. Each module is specified as an abstract machine after which each abstract machine is implemented in terms of the data objects and the functions provided by the level beneath it.

Although HDM includes both specification and verification, the specification phases have been much further developed and tested than the verification phases. Module specifications are written in a nonprocedural language, Special (SPECIfication and Assertion Language). Starting with a nonprocedural specification helps to decompose the verification process; properties which are independent of implementation can be proven before implementation decisions are introduced which increase the complexity.

SRI has been active in the area of verification but up till now effort has mostly concentrated on isolated areas. An effort is currently underway at SRI to provide an integrated specification, implementation and verification environment; a necessary effort if HDM is to be used on a system from system conception through life-cycle maintenance

HDM has a history of use, most of it in the area of systems requiring security. Much of its original development occurred during an SRI project to design a Provably Secure Operating System (PSOS) [Neu77]. It has also been used in a good number of the security kernel systems which have been developed over the past few years. Of the projects

using HDM the effort to build a Kernelized Secure Operating System (KSOS) is the farthest into development. As part of the KSOS project a formal description of the security kernel was written for both the kernel interface and the levels into which the kernel is decomposed [FAC79]. Since an emulator for the UNIX operating system was to be provided to run in the kernel, a formal specification of the interface of the UNIX operating system was also written [BD78].

3. Affirm

Affirm is an interactive verification system under development at the University of Southern Califirnia's Information Sciences Institute (ISI). It has grown out of two earlier verification projects, the XIVUS Pascal verifier and the Data Type Verification System. It can be used to prove properties about programs using abstract data types and verify implementations which provide said properties [Tho79].

Like the Gypsy system, Affirm relies on the general notion of refinement as its development methodology. It too is amenable to being used to specify a hierarchy of abstract machines.

Specifications for the Affirm system are written in an algebraic form based on predicate calculus; programs are implemented in a variant of Pascal.

AFFIRM has been used on a varying collection of problems. The DELTA experiment [GW79] involved specifying and verifying part of a message processing system. It was used to verify parts of the UCLA security kernel [Kem79], but it was not possible to verify the complete

kernel (2). There has also been some work in using AFFIRM to state properties about protocols [Sun80].

4. Stanford Pascal Verifier

The Stanford Pascal Verifier (SPV) is an interactive system for analyzing a program for consistency with its specification [Luc79]. It is primarily a system for program verification. It contains no support for abstract data type verification or hierarchical specification development.

Specifications for the SPV system are written in an algebraic form based on predicate calculus; programs are implemented in a variant of Pascal called Pascal Plus. The SPV is somewhat similar to the AFFIRM system - it is stronger in the area of program verification and weaker in its support of higher level constructs.

The SPV has mostly been used on small examples and has been used by classes at Stanford University on program verification. The most significant program, in terms of size, verified by SPV has been a compiler for a Pascal-like language [Pol80].

c. Hierarchical Development Methodology

Under the criteria set forth for choosing a methodology the Hierarchical Development Methodology seemed to be the most reasonable choice as a framework for developing protocols. It is a

---

(2) The author has used all of the systems under consideration; it is doubtful any of them could have verified the complete kernel.

development methodology which encompasses design, specification, implementation, and verification of large software systems and a methodology specifically aimed at programs which are amenable to being conceptually viewed as a hierarchy of abstract machines.

The methodology decomposes the system design into stages and involves design, implementation and verification. The implementation stages not only enforce good design and coding practices but also are necessary to meet the verification goals. The verification stages prove that the design meets the specifications as it evolves and that the implementation programs correctly implement the system as specified. The design stages and their corresponding verification stages are shown in figure 3 [Rob79].

Design	Verification
s0: Conceptualization of system requirements	v0: develop global assertions
s1: Selection of user interface and target machine	
s2: decompose system into hierarchical levels	
s3: represent the functions of each level as modules	v3: prove the modules satisfy global assertions
s4: map state information between levels	v4: show the mappings are consistent
s5: implement the functions of level n in terms of the functions at level n-1	v5: prove the abstract implementation consistent with module specifications
s6: Conversion of the abstract implementations to executable code	v6: Prove the executable code consistent with the abstract implementations

Fig. 3

HDM is a mix between an axiomatic and an abstract machine approach. The first stage calls for the development of global assertions (axioms), then at a later stage abstract machines are developed which, as a minimum, are consistent with these assertions. In development efforts where HDM has been used this two tiered approach has been used to allow well understood characteristics of the system to be axiomatized (e.g. the security model for the KSOS security kernel) and the less understood characteristics to be given by the abstract machine specification (e.g. describing the interface of the KSOS kernel). We will continue this practice in the specifications below.

Perhaps the greatest concern about the use of HDM arose not in the area of the methodology itself, but with the specification language, Special, associated with it. Special is known to have some deficiencies in the area of concurrent processes. As communication protocols will require a specification language which has a good grasp of concurrency, this could prove to be a serious problem.

The possibility existed of using another specification language in place of Special. Perhaps the best choice in this area would have been from the work of the Certifiable Minicomputer Project (CMP) at the University of Texas at Austin. CMP has developed a methodology for specifying, implementing and verifying communications processing software with particular emphasis on the problems of concurrency [Goo77]. This system is not based on using a hierarchy of abstract machines, but, according to its designers, can be used in conjunction with a methodology using a hierarchy of abstract machines.

The specification language for the system, GYPSY [Goo78], would be well suited for the task of protocol specification as its model of concurrent processes involves communication through message buffers. The major problem that occurs is that, although tools exist to support the user for either system, these tools are not compatible. Since this is intended as an exercise in specification rather than tool building it was decided to stick with the language of HDM since, after examining the exact problems, it seemed that Special would be adequate for the specifications considered to be needed for this thesis. This assessment was not entirely accurate, as will be described within the specifications and conclusions.

An interesting exercise to pursue in the future would be to specify the Internet Protocol (the example used below) in GYPSY and compare the utility of the two approaches. Some comparison studies using simple protocols have been made by Sunshine [Sun79].

One definite variance from HDM will be necessary for specifications in the area of protocols. As protocols, by their very nature, are intended for multiple systems, it will not be possible to define the target machine except in the special case of a network using the same target machine throughout. Instead each implementation will have to build up from its target machine the necessary functions to implement the protocol. Therefore, one goal of a protocol specification will be machine independence.

d. SPECIAL

The specification language in which the modules are written is called Special (SPECIfication and Assertion Language). It is a nonprocedural specification language which formalizes a technique set forth by Parnas [Par72]. Detailed information about Special can be found in the language reference manual [RR77], a handbook on the use of the languages of HDM [SLR 79], and the formal mathematical description of the language [BM 78]. Of these the handbook gives the best insight into why the specifications take the form they do. The paragraphs below describe the major features of Special used in this thesis. Uncommon features will be described within the actual specifications as comments when used. Special keywords are set in capital letters in the description.

A module consists of six sections termed paragraphs: TYPES, PARAMETERS DEFINITIONS, EXTERNALREFS, ASSERTIONS, and FUNCTIONS. Any empty paragraph is omitted.

Special is a strongly typed language and there are four basic (built-in) types: INTEGER, BOOLEAN, CHAR(acter), and DESIGNATOR. The designator type is used to give objects unique names. An implementation must require that no two objects with identical designators exist simultaneously. Any objects which do not consist of these predefined types must be built from the basic types in the TYPES paragraph of the module. The major composing operators are VECTOR~OF, SET~OF, and STRUCT~OF. Additionally new types which are simply one of the basic types may be declared to signify that two objects which are of one of

the basic types are not interchangeable.

The PARAMETERS paragraph is used to declare objects which remain fixed over any given instantiation of the module. This allows implementation dependent information to be represented in the specification without being overly restrictive.

The DEFINITIONS paragraph essentially provides global macro expansions for the module. In the specification herein it is used to give names to error (EXCEPTION) conditions which arise within the module and names to some non-primitive constants of the module.

The EXTERNALREFS paragraph allows a module to import functions, types, and exception conditions from another module.

The ASSERTIONS paragraph gives global properties maintained by the module. They are invariants which are to be proven as part of the verification.

The FUNCTIONS paragraph defines the state of the machine and the available operations on the machine. There are two basic function types in Special. A V-function (VFUN) is a function which contains and returns information about the state of the abstract machine. An O-function (OFUN) changes the state of the abstract machine. Additionally there is an OV-function (OVFUN) for operations which would be critical sections in the actual code; it both maintains and changes state information. Functions may have an EXCEPTIONS section which consists of a set of boolean expressions. If any exception evaluates to true the exception is returned; no other action occurs.

A VFUN may be either HIDDEN or visible. Visible VFUNs specify the state of the abstract machine which can be observed from a higher level abstract machine. HIDDEN VFUNs contain information used by the abstract machine to implement its functionality, but are not part of the interface (i. e. information hiding). HIDDEN VFUNs, since they are for internal use only, have no EXCEPTIONS; the functions which call the HIDDEN function must be verified to do all the necessary exception checking.

A VFUN may also be either primitive or derived. The state of an abstract machine is represented by its primitive VFUNs. Only primitive VFUNs may be cited in the EFFECTS of an OFUN. Derived VFUNs exist to present the state of the machine in a way more useful to the next higher level.

OFUNs are used to define the operations on the data structures of the module. Like visible VFUNs, they have EXCEPTIONS and the interpretation is identical. The result of calling the OFUN is stated in the EFFECTS section. All EFFECTS are mathematical statements; assignment is never implied. It defines a new, but not necessarily unique, state for the abstract machine in terms of the current state. Nondeterminism in the specification is encouraged both to point out areas where the requirements may need to be made more restrictive and to give the implementor as much freedom as possible so that the most efficient implementation which meets the requirements is allowed.

A few notational remarks:

a) A new value of a VFUN (assigned by an OFUN) is signified by preceding the name by a single quote (').

b) A VFUN designated as HIDDEN is only available to the module in which it is defined; it may not be accessed by other modules.

c) Question mark symbol (?) is a distinguished value signifying UNDEFINED. A statement of the form "<var> = ?;" in the INITIALLY section of a VFUN implies that for any set of arguments to the VFUN for which no value has been defined by some call to an OFUN, the value of the VFUN is the distinguished value, UNDEFINED.

d) The statement *RESOURCE~ERROR in an exception section* implies that an error occurred in a lower level module implementing the function. It allows lower level resource allocation decisions to be concealed.

e) The NEW operator applies to an object of type DESIGNATOR. It returns a unique DESIGNATOR each time it is called and therefore can be used to distinguish objects.

f) The construct

VECTOR~OF a | LENGTH(a) = b

is read as defining a vector of objects of type a, the vector being b long where b must be of type INTEGER.

g) The construct ~= symbolizes not equal.

3. A HIERARCHY OF PROTOCOLS

Communication protocols, as stated above, are developed in layers. These layers are each meant to present an abstract machine interface for higher level protocols, both known and to be developed in the future. This differs somewhat from the development of operating systems in which a single visible interface is presented.

Figure 2 above presented protocol layering in the abstract; figure 4 presents a part of an existing protocol hierarchy. (The levels beside the figure refer to figure 2.) This is a small selection from the hierarchy which exists and is planned for the ARPA internetworking experiment. Of particular note is that not only is a lower level protocol expected to support multiple higher level protocols, but also that higher levels may be expected to be implemented on different lower level protocols.

At the highest level of specification, each protocol can be viewed as providing a user interface for a specific type of service and mapping the input parameters into a call to the next lower level which will achieve the desired result.

Specifying the user interface will not lead to a single view of the system but a view at each layer because each protocol is meant to be a "user interface" to the next higher level and to allow for additional protocols to be implemented at that level.

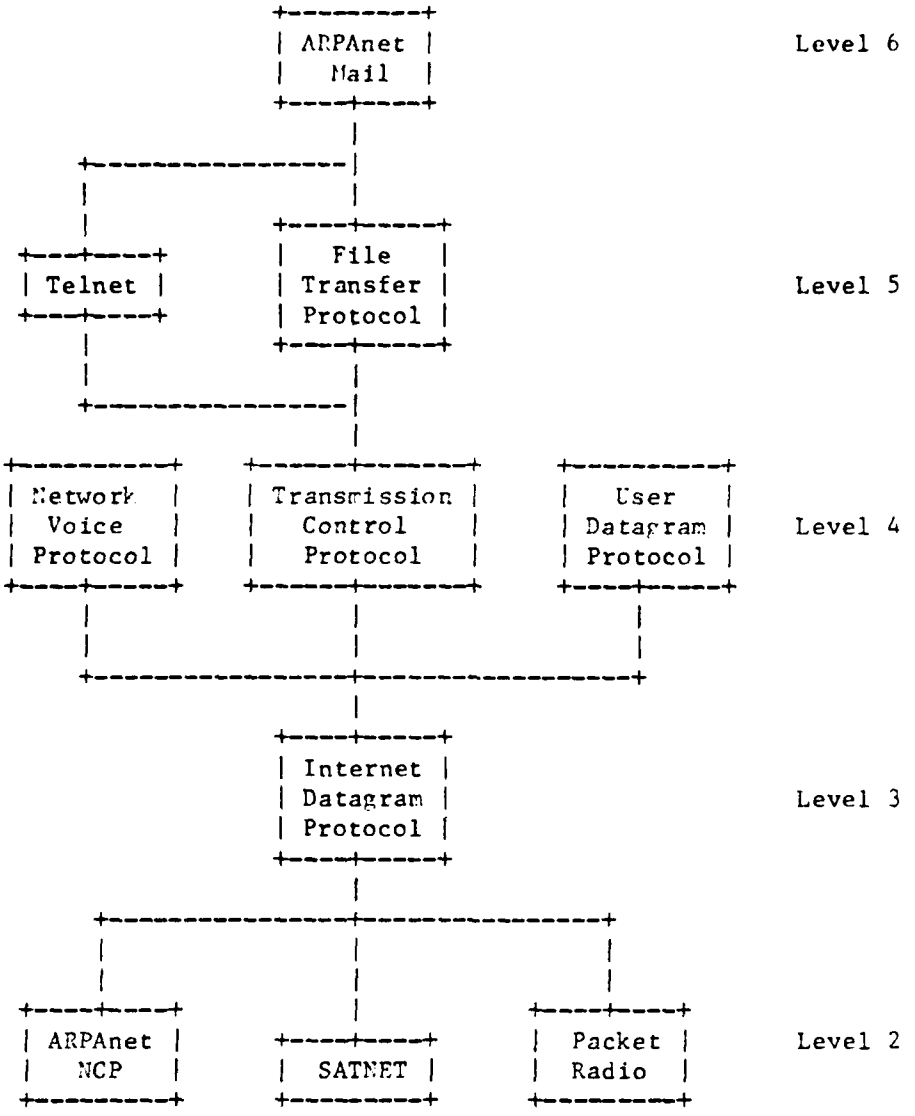


Fig. 4

In figure 4 each protocol is expected to run using the functions and data objects provided by the protocol layer beneath it. In this model a layer can be viewed as an abstract machine. Often the layering is violated; sometimes minimally, sometimes seriously. No matter how minimal the violation this "reaching into" another layer violates the

abstract machine concept and intertwines the protocols. This intermixing leads to the problems mentioned in the introduction -- lateness, complexity, etc.

A protocol has two different kinds of specification -- an interface specification and an internal specification (figure 5). The interface specification presents the abstract machine interface provided by the protocol. The internal specification gives the details of the abstract machine's internal structure. The internal specification presents the necessary information to build the interface specification from the functions provided by the abstract machine on which the protocol under consideration is to run.

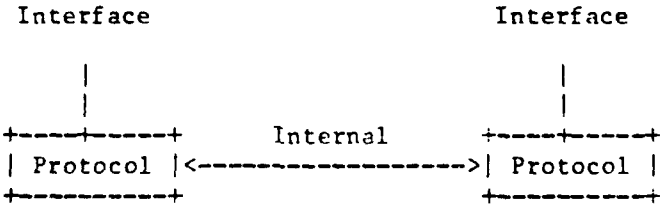


Fig. 5

The distinction drawn above, though artificial in the sense that all of the layers are composed of abstract machines, tends to emphasize the orthogonal purposes which the specifications serve.

Communication protocols are often believed to present some verification problems which are significantly different from those found in operating systems and data base systems. Because communication protocols require cooperating processes across rather

error-prone transmission media, one must worry about corruption and loss of data. The difference, from the viewpoint of specification and verification, is mostly one of degree. Operating systems and data base systems are subject to losses due to hardware failure and the inputs received from an unspecified user process are probably more random than from most transmission media. Probably the most important difference is that since the characteristics of a communication medium are predictable (in a probabilistic sense) more can be said about various performance considerations of protocols.

The specification and verification problems mentioned above are serious issues and research is needed into verification techniques for such problems. This is not meant to imply a need for a separate area called "protocol verification" because these problems need to be solved for operating systems and data base systems. The need is to extend current techniques in the area of program verification.

There is still though, one obvious difference which makes communication protocol software unique and this difference cannot be overlooked -- by its very nature communication protocol software is distributed among at least two physically separated entities. The effect this has on the specification process will be a topic of discussion later.

a. Choosing a protocol

The discussion above gives a framework for specifying protocols. It requires only that a protocol be pictured as an abstract machine.

These abstract machines are arranged hierarchically providing a more concrete view as the level of refinement increases. The next step is to pick a point in the hierarchy to begin the specification task.

Of the possible points to begin the specification process in the hierarchy outlined above the internet layer seems to be a logical candidate. Although this may seem to be a middle out approach, this layer seems to be the fundamental layer to which protocols above it and below it must interface. The networking model we have given does not state how many distinct protocols may exist at each layer. At most layers many different protocols already exist. At the internet layer, however, the choice is a single protocol, adding a further layer (i.e. an inter-internet layer) or a collection of ad hoc connections.

Currently there are three major protocols being proposed for network interconnection. The Defense Advanced Research Projects Agency (DARPA) Internet Protocol (IP) [Pos80a], the PUP architecture of Xerox Palo Alto Research Center (PARC) [Bog80], and the Consultative Committee on International Telephone and Telegraphy (CCITT) X.75 recommendation [X.75]. Both IP and PUP are datagram oriented; X.75 is virtual circuit oriented.

Although all three of these protocols are intended for network interconnection, they do not approach the problem at the same network layer. The IP and PUP proposals provide only end-to-end addressing; higher level protocols are responsible for reliability. The X.75 recommendation is intended to provide reliable end to end delivery. In this sense PUP and IP are at level 3 in the protocol hierarchy (see

figure 4) whereas X.75 is at level 4. Being at a higher level X.75 is also the least general -- it is designed to connect only networks which use the CCITT X.25 protocol as their network access protocol.

The differences between IP and PUP are mostly matters of philosophy. Of the two PUP tends to be simpler, mostly because it does not handle fragmentation - the splitting of datagrams which are too large to fit into another network. PUP does not ignore this problem, it just considers it to be a problem to be handled by the gateways into and out of each network. IP opted for simpler gateways with a more complex internetworking protocol. Another difference is IP does addressing on a host to host basis whereas PUP does addressing on a process to process basis.

Although PUP and IP may differ in some minor matters of philosophy the designers of both agree on one major point - there needs to be a single internet layer.

We have chosen to study the IP in this thesis for two reasons. It has been established as a standard for the Department of Defense. Also, an agreement between the ARPAnet and PUP communities [Coh79] states that when a conflict arises between IP and PUP as an internetwork protocol PUP will be encapsulated in IP; therefore, we decided to use IP as an example in this thesis.

b. Distributing modules among network entities

HDM has as one of its goals to be a methodology applicable to designing software systems which in some sense constitute a family of processes. SRI has investigated this to a degree in an effort to define a family of operating system for the Army but this effort was discontinued [Neu76]. This work forms a basis for using HDM with a system which is a family of processes such as the protocol model given in figure 4. The effect of distributing the modules of a family among the loosely coupled systems of a network needs to be investigated.

The earily stages of HDM (0-2) purposefully attempt to place minimal restrictions on implementation decisions; the only requirement is that the implementation meet the specification. Modules may be implemented as hardware, software, firmware or a combination thereof; they may run on the same processor or be distributed among many processors. SRI has an effort underway which involves distributed processing. The SIFT (Software Implemented Fault Tolerance [Wen76]) system is a collection of distributed processors which use software redundancy among processors to compensate for hardware failures. Although this effort will face some of the problems which a distributed network raises, it is not as severe a test as a system such as the ARPAnet because the whole effort is being done at one location by a single, closely coordinated team. In networks there is a need to communicate implementation decision which must be made in common by all processors to allow effective communication.

The major problem which independent implementations seems to cause is that the specification must include some information which is normally not needed because the compiler hides such details. A compiler will use internal representations for objects which are consistent across module boundaries. If modules are compiled using different compilers (a very likely circumstance in a heterogeneous network), either the compilers must use a common representation when objects from one module need to be interpreted by another or they must have the ability to distinguish and transform from one representation to another.

4. THE INTERNET PROTOCOL

One item any formal specification needs is a commentary explaining the overall purpose of each major subsystem and the philosophy behind its specifications. The commentary should reveal to the reader the decisions behind the mathematical description. The goal is not to bias the reader but to guide him, explaining to him the purpose behind each function in the specification. Recall the GCD function given above. Without being told what function was being represented some one unfamiliar with this set of axioms may have to try many possible functions before concluding that the axioms define the GCD function. Even though a commentary will only give an informal notion of the system being specified, this can greatly enhance the speed of comprehension. There is a danger with providing such a commentary. It is much easier to decide (mistakenly) that a function performs a specific function after being told the intended function than to

recreate the original mistake. It is the reader's job to confirm that the specification corresponds to both his understanding of the purpose of the system under development and the informal description. In the specifications which follow, specific descriptive information about each unit within the specification will be given as comments within that unit.

A good description of the IP already exists [Pos80a]. Excerpts relevant to understanding the formal specifications (3) are included below.

There are three parts of the description in which we are interested: the purpose, the interface description and the internal description. The purpose is essentially the conceptualization of the system requirements, stage s0 of HDM. The purpose provides the basis for the global assertions of the system, stage v0. This stage requires that the informal English statement of the purpose be restated in formal mathematics terms. The s1 stage of HDM requires choosing two interfaces -- the interface the IP will present to users and the interface of the abstract machine on which the IP will run. The protocol touches on both of these matter but it concentrates mostly on the internal description of the protocol. The internal description is fairly complete, although the actions to be taken in the event of some unexpected situations are often undefined.

---

(3) Henceforth the term protocol description will be used to refer to [Pos80a]. The term protocol specification will not be so used.

a. Purpose

> The Internet Protocol is designed for use in interconnected systems of packet-switched computer communication networks. The internet protocol provides for transmitting blocks of data called datagrams from sources to destinations, where sources and destinations are hosts identified by fixed length addresses. The internet protocol also provides for fragmentation and reassembly of long datagrams, if necessary, for transmission through "small packet" networks.
>
> The internet protocol is specifically limited in scope to provide the functions necessary to deliver a package of bits (an internet datagram) from a source to a destination over an interconnected system of networks. There are no mechanisms to promote data reliability, flow control, sequencing, or other services commonly found in host-to-host protocols.

The above description gives the major purpose the IP is intended to fulfill. Although nowhere explicitly stated in the IP description, there are a number of other services which the IP is expected to provide. These are provided by the IP as what are termed internet options.

Though an informal statement of the requirements is rather easy, mathematical formalization is much harder. The state of formal specification is not at a point where all requirements can be stated mathematically in such a manner so as to be proven.

We can state some properties about the IP which should be provable. These properties are rather weak as the service provided by the IP has very little sophistication. The IP does not specifically concern itself with anything but the correctness of the internet header. It provides a checksum for the header but none for the data. Any strong correctness criteria will only be able to deal with the contents of the internet header.

Despite the fact that the IP makes no claims about the correctness of the data it delivers, the IP must have some concern for it. Although the description never explicitly states it, any implementation of the IP which always delivered corrupted datagrams would have to be considered incorrect. Informally it can be assumed that an implementor knows this but a formal specification would need to include a statement to the effect that no deterministic step will ever cause a datagram to be delivered in error, or that if sent often enough a datagram will get through with the data intact or recoverable (i. e. successful forward error correction).

b. Interface description

Because of the generality of the higher level protocols which the IP is meant to service, it essentially presents a set of user interfaces. These interfaces are meant to accommodate users whose requirements range from needing nothing but an unordered stream of datagrams to those wanting to implement highly structured data streams. The protocol description gives two sample calls as an example interface [Pos80a].

> The following two calls satisfy the requirements for the user to internet protocol module communication ("=>" means returns):
>
> SEND (dest, TOS, TTL, BufPTR, len, Id, DF, options => result)
>
> where:
>
> dest = destination address
> TOS = type of service
> TTL = time to live
> BufPTR = buffer pointer
> len = length of buffer
> Id = Identifier

DF = Don't Fragment
options = option data
result = response
OK = datagram sent ok
Error = error in arguments or local network error

RECV (BufPTR => result, source, dest, prot, TOS, len)

where:

BufPTR = buffer pointer
result = response
OK = datagram received ok
Error = error in arguments
source = source address
dest = destination address
prot = protocol
TOS = type of service
len = length of buffer

In the course of specifying the protocol two errors were found in the sample RECV call. One problem is that the call should contain a return parameter for the IP options. However, not all option types should be visible at the interface. The NO OP and end of options option are implementation details and the General Error Report is an exceptions reporting mechanism. (The types of options are described in detail below.) The second problem is that the protocol should be a parameter to the call, not a result. With these emendations the call would be:

RECV (BufPTR, prot => result, source, dest, TOS, opt, len)

c. Internal description

The internal description essentially defines an abstract data type called an internet datagram or packet. Having this well defined format solves many of the problems referred to in the section discussing what the consequences are of spreading IP implementation among heterogeneous

computers on a network.

The format of the IP header is given in figure 6. It, and the description following, are drawn from [Pos80a]; the footnotes are not from the quoted source.

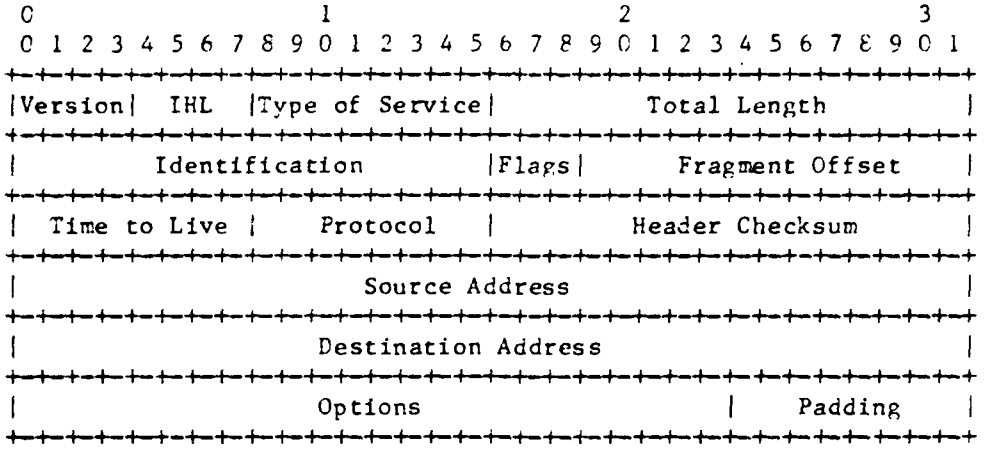


Fig. 6

Each tick mark represents one bit position.

Version: 4 bits

The Version field indicates the format of the internet header. This document describes version 4.

IHL: 4 bits

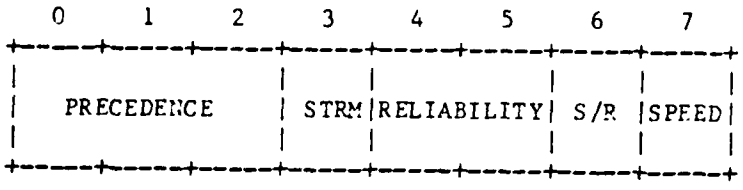
Internet Header Length is the length of the internet header in 32 bit words, and thus points to the beginning of the data. Note that the minimum value for a correct header is 5.

Type of Service: 8 bits

The Type of Service provides an indication of the abstract parameters of the quality of service desired. These parameters are to be used to guide the selection of the actual service parameters when transmitting a datagram through a particular network.

Bits 0-2: Precedence.
Bit 3: Stream or Datagram.
Bits 4-5: Reliability.

Bit 6: Speed over Reliability.
Bits 7: Speed.



PRECEDENCE	STRM	RELIABILITY	S/R	SPEED
111-Flash Override	1-STREAM	11-highest	1-speed	1-high
110-Flash	0-DTGRM	10-higher	0-rlblt	0-low
11X-Immediate		01-lower		
01X-Priority		00-lowest		
00X-Routine				

The type of service is used to specify the treatment of the datagram during its transmission through the internet system.

Total Length: 16 bits

Total Length is the length of the datagram, measured in octets, including internet header and data. This field allows the length of a datagram to be up to 65,535 octets. All hosts must be prepared to accept datagrams of up to 576 octets (whether they arrive whole or in fragments).

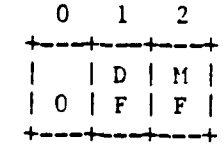
Identification: 16 bits

An identifying value assigned by the sender to aid in assembling the fragments of a datagram.

Flags: 3 bits

Various Control Flags.

Bit 0: reserved, must be zero
Bit 1: Don't Fragment This Datagram (DF).
Bit 2: More Fragments Flag (MF).



Fragment Offset: 13 bits

This field indicates where in the datagram this fragment belongs. The fragment offset is measured in units of 8 octets (64 bits). The first fragment has offset zero.

Time to Live: 8 bits

This field indicates the maximum time the datagram is allowed to remain in the internet system. If this field contains the value zero, then the datagram should be destroyed. This field is modified in internet header processing. The time is measured in units of seconds. The intention is to cause undeliverable datagrams to be discarded.

Protocol: 8 bits

This field indicates the next level protocol used in the data portion of the internet datagram.

Header Checksum: 16 bits

A checksum on the header only. Since some header fields may change (e.g., time to live), this is recomputed and verified at each point that the internet header is processed.

The checksum algorithm is:

The checksum field is the 16 bit one's complement of the one's complement sum of all 16 bit words in the header. For purposes of computing the checksum, the value of the checksum field is zero.

Source Address: 32 bits

The source address. The first octet is the Source Network, and the following three octets are the Source Local Address.

Destination Address: 32 bits

The destination address. The first octet is the Destination Network, and the following three octets are the Destination Local Address.

Options: variable

The option field is variable in length. There may be zero or more options. There are two cases for the format of an option:

Case 1: A single octet of option-type.

Case 2: An option-type octet, an option-length octet, and the actual option-data octets.

The option-length octet counts the option-type octet and the option-length octet as well as the option-data octets.

The option-type octet is viewed as having 3 fields:

1 bit reserved, must be zero
2 bits option class,
5 bits option number.

The option classes are:

0 = control
1 = internet error
2 = experimental debugging and measurement
3 = reserved for future use

The following internet options are defined:

CLASS	NUMBER	LENGTH	DESCRIPTION
0	0	-	End of Option list. This option occupies only 1 octet; it has no length octet.
0	1	-	No Operation. This option occupies only 1 octet; it has no length octet.
0	2	4	Security. Used to carry Security, and user group (TCC) information compatible with DOD requirements.
0	3	var.	Source Routing. Used to route the internet datagram based on information supplied by the source.
0	7	var.	Return Route. Used to record the route an internet datagram takes.
0	8	4	Stream ID. Used to carry the stream identifier.
1	1	var.	General Error Report. Used to report errors in internet datagram processing.
2	4	6	Internet Timestamp.
2	5	6	Satellite Timestamp.

Specific Option Definitions

End of Option List

Type=0

This option indicates the end of the option list. This might not coincide with the end of the internet header according to the internet header length. This is used at the end of all options, not the end of each option, and need only be used if the end of the options would not otherwise coincide with the end of the internet header.

May be copied, introduced, or deleted on fragmentation.

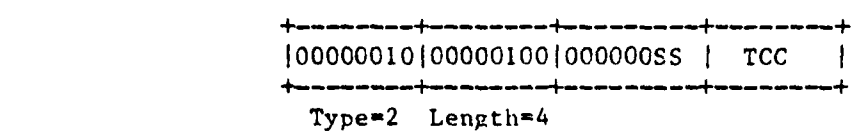
No Operation

This option may be used between options, for example, to align the beginning of a subsequent option on a 32 bit boundary.

May be copied, introduced, or deleted on fragmentation.

Security

This option provides a way for DOD hosts to send security and TCC (closed user groups) parameters through networks whose transport leader does not contain fields for this information. The format for this option is as follows:



Security: 2 bits

Specifies one of 4 levels of security

11-top secret
10-secret
01-confidential
00-unclassified

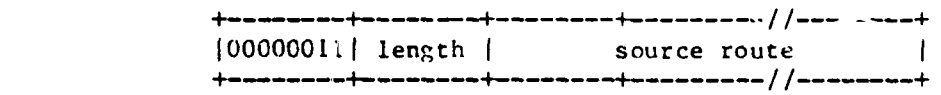
Transmission Control Code: 8 bits

Provides a means to compartmentalize traffic and define controlled communities of interest among subscribers.

Note that this option does not require processing by the internet module but does require that this information be passed to higher level protocol modules. The security and TCC information might be used to supply class level and compartment information for transmitting datagrams into or through AUTODIN II.

Must be copied on fragmentation.

Source Route



Type=3

The source route option provides a means for the source of an internet datagram to supply routing information to be used by the gateways in forwarding the datagram to the destination.

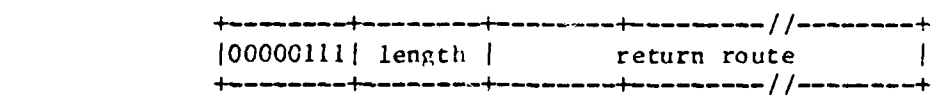
The option begins with the option type code. The second octet is the option length which includes the option type code and the length octet, as well as length-2 octets of source route data.

A source route is composed of a series of internet addresses. Each internet address is 32 bits or 4 octets. The length defaults to two, which indicates the source route is empty and the remaining routing is to be based on the destination address field.

If the address in destination address field has been reached and this option's length is not two, the next address in the source route replaces the address in the destination address field, and is deleted from the source route and this option's length is reduced by four. (The Internet Header Length Field must be changed also.)

Must be copied on fragmentation.

Return Route



Type=7

The return route option provides a means to record the route of an internet datagram.

The option begins with the option type code. The second octet is the option length which includes the option type code and the length octet, as well as length-2 octets of return route data.

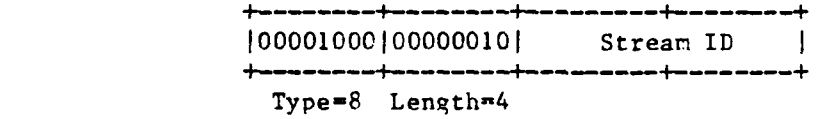
A return route is composed of a series of internet addresses. The length defaults to two, which indicates the return route is empty.

When an internet module routes a datagram it checks to see

if the return route option is present. If it is, it inserts its own internet address as known in the environment into which this datagram is being forwarded into the return route at the front of the address string and increments the length by four.

Not copied on fragmentation, goes in first fragment only.

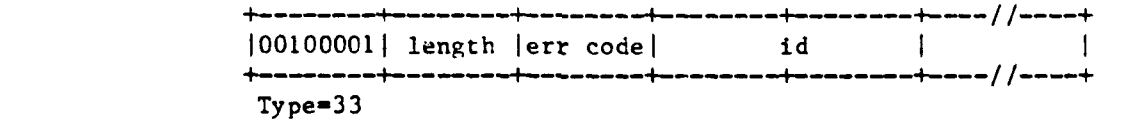
Stream Identifier



This option provides a way for the 16-bit SATNET stream identifier to be carried through networks that do not support the stream concept.

Must be copied on fragmentation.

General Error Report



The general error report is used to report an error detected in processing an internet datagram to the source internet module of that datagram. The "err code" indicates the type of error detected, and the "id" is copied from the identification field of the datagram in error, additional octets of error information may be present depending on the err code.

If an internet datagram containing the general error report option is found to be in error it must be discarded, no error report is sent.

ERR CODE:

0 - Undetermined Error, used when no information is available about the type of error or the error does not fit any defined class. Following the id should be as much of the datagram (starting with the internet header) as fits in the option space.

1 - Datagram Discarded, used when specific information is available about the reason for discarding the datagram can be reported. Following the id should be the original (4-octets) destination address, and the (1-octet) reason.

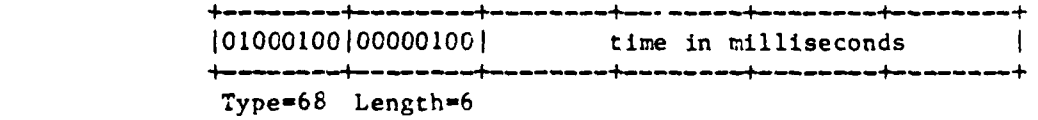
Reason	Description
0	No Reason
1	No One Wants It - No higher level protocol or application program at destination wants this datagram.
2	Fragmentation Needed & DF - Cannot deliver with out fragmenting and has don't fragment bit set.
3	Reassembly Problem - Destination could not reassemble due to missing fragments when time to live expired.
4	Gateway Congestion - Gateway discarded datagram due to congestion.

The error report is placed in a datagram with the following values in the internet header fields:

Version: Same as the datagram in error.
IHL: As computed.
Type of Service: Zero.
Total Length: As computed.
Identification: A new identification is selected.
Flags: Zero.
Fragment Offset: Zero.
Time to Live: Sixty.
Protocol: Same as the datagram in error.
Header Checksum: As computed.
Source Address: Address of the error reporting module.
Destination Address: Source address of the datagram in error.
Options: The General Error Report Option.
Padding: As needed.

Not copied on fragmentation, goes in first fragment only (4).

Internet Timestamp



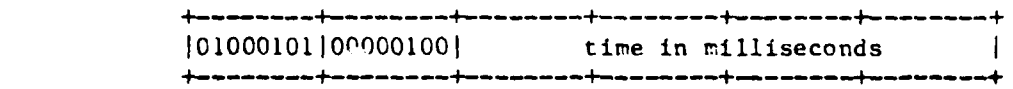
The data of the timestamp is a 32 bit time measured in milliseconds.

Not copied on fragmentation, goes in first fragment only.

---

(4) This statement is correct, however, a GER cannot be fragmented because it is only a datagram header.

Satellite Timestamp



The data of the timestamp is a 32 bit time measured in milliseconds.

Not copied on fragmentation, goes in first fragment only.

Padding: variable

The internet header padding is used to ensure that the internet header ends on a 32 bit boundary. The padding is zero (5).

d. Interface vs. internal description

Although the IP description is intended to be both an interface and an internal description of the protocol it is more concerned with describing the internal structure of an IP implementation than with presenting a good interface to the IP. Consider the way the description treats the the routing options of IP.

SEND(sa A, da B, sr <C, D, E>, rr <>)

sa - source address

da - destination address

sr - source route

rr - return route

capital letters are internet addresses

angle brackets are used to enclose a list of addresses

The interpretation of this call would be to deliver a datagram to E

---

(5) Effectively end of option list bytes.

passing through B, C and D on the way. When it arrived at E a RECV call would return

(sa A, da E, sr <>, rr <D, C, B>)

(This makes the assumption that the only internet routing along the way is done at B, C and D). To return the packet to A the next level must reformat these arguments as follows:

(sa E, da D, sr <C, B, A>)

In addition to being slightly confusing using this format makes the assertions about the IP more complicated. Instead of being able to state that if a datagram is delivered on a RECV call that a SEND call was made with this address as the destination, it is necessary to include a special clause to deal with the way routing options work.

The problems could be "solved" by defining the call at the interface differently. Let the call at the interface be

SEND(sa A, da E, sr <B, C, D>, rr <>)

and have the IP reformat the arguments internally as they were in the call above - (sa A, da B, sr <C, D, E>, rr)

At the receiving end the argument returned by RECV can be used to return an answer simply by swapping source and destination. This reformatting also removes the need to have a special case in the assertion about where the datagram will be delivered.

The situation described above normally occurs due to the way design is done. The requirement stated would be to the effect that the protocol provide a way to do explicit routing and to collect routing information on the way. No further refinement of the requirement is often done until implementation decisions are made. This decision then becomes the interface to the system.

The requirements should not overconstrain the implementation. However, after implementation decisions are made which reflect upward, the upper level specification should be revised to present a clear, consistent interface.

5. USER INTERFACE FORMAL SPECIFICATIONS

The interface described here is different from the interface which is necessary to use the full functionality provided by IP. It attempts to provide an interface which hides all of the implementation specific details of the IP. In many ways it is the type of interface which would be very appropriate to the User Datagram Protocol (UDP) [Pos79]. Most of this interface specification could be directly imported into a specification for the UDP. It would not require any changes, just the addition of four fields of the UDP header which would map on to the data portion of the IP.

a. Module: Inet~send

This module provides the user interface to the IP. It is intended to be an interface which hides all inner workings of the implementation

from the user. Certain parameters defined in the protocol description are not available at this interface (e. g. the identification field). Higher level protocols which require access to such fields must have access to the abstract machine which implements Inet~send. The VFUNs of this module are all HIDDEN as the protocol description has no functions which return this information. If desired a derived VFUN, Status, could be defined.

```
MODULE Inet~send

TYPES

     Byte: {0..2~Byte~size-1};
     Flag: BOOLEAN;
     Data: VECTOR~OF Byte;
     Char~str: VECTOR~OF CHAR;
     Optn: STRUCT~OF(Char~str opt~type, INTEGER opt~ln;
                     Data opt~info);
     $(At the user level options consisting of a single octet
     should never be visible.)
     Options: SET~OF Optn;
     Pkt~id: DESIGNATOR;
     Type~of~serv: STRUCT~OF(CHAR precedence; Flag strm;
                      CHAR reliability; Flag svsr, speed);
     Packet: STRUCT~OF(INTEGER ver; Type~of~serv tos; INTEGER ttl;
             Char~str pcol, sa, da; Options o; Data d);

PARAMETERS

     INTEGER Version;
     INTEGER Byte~size;
     INTEGER Source~net;
     SET~OF Char~str Source~addrs;
     Char~str Default~source~addr;
     INTEGER Max~inet~pkt;
     SET~OF Char~str Legal~addr;
     SET~OF Char~str Option~types;
     $(This set would not contain NO OP, End of Options, or GER.)
     SET~OF Char~str Protocols;
     SET~OF Char~str Service~types;
```

```
DEFINITIONS

     BOOLEAN Unknown~addr(Char~str addr) IS
             NOT (addr INSET Legal~addr);

     BOOLEAN Unknown~pcol(Char~str c) IS NOT(c INSET Protocols);

     BOOLEAN Unknown~opt(Optn opt) IS NOT(opt.opt~type INSET
             Option~types);

     BOOLEAN Invalid~tos(Char~str c) IS NOT(c INSET Service~types);

     BOOLEAN Invalid~pkt(Pkt~id id) IS NOT(id INSET Current~ids());

     BOOLEAN Too~small(INTEGER t) IS t <= 0;


EXTERNALREFS

     FROM Virt~net:
        OFUN Send~to~net(Packet p);


FUNCTIONS

VFUN Type~of~service(Pkt~id id) -> Type~of~serv tos;
     HIDDEN;
     INITIALLY tos = ?;

VFUN Time~to~live(Pkt~id id) -> INTEGER ttl;
     $(Time measured in seconds. The General Error Report has a
     default time to live of sixty seconds. This type of packet
     does not appear at this interface, however, so the default
     of UNDEFINED will be used for all cases which occur at the
     user interface.)

     HIDDEN;
     INITIALLY ttl = ?;

VFUN Protocol(Pkt~id id) -> Char~str pcol;
     HIDDEN;
     INITIALLY pcol = ?;

VFUN Source~addr(Pkt~id id) -> Char~str sa;
     HIDDEN;
     INITIALLY sa = Default~source~addr;

VFUN Dest~addr(Pkt~id id) -> Char~str da;
     HIDDEN;
     INITIALLY da = ?;

VFUN Option~field(Pkt~id id) -> Options opt;
```

```
     HIDDEN;
     INITIALLY opt = { };

VFUN Msg(Pkt~id id) -> Data m;
     HIDDEN;
     INITIALLY m = ?;

VFUN Packets~in~prog(Pkt~id id) -> Packet p;
     EXCEPTIONS
        Invalid~pkt(id);

     DERIVATION
        STRUCT (Version,
        Type~of~service(id),
        Time~to~live(id),
        Protocol(id),
        Source~addr(id),
        Dest~addr(id),
        Option~field(id),
        Msg(id));

VFUN Current~ids() -> SET~OF Pkt~id ids;
     HIDDEN;
     INITIALLY ids = { };

OVFUN Make~pkt() -> Pkt~id id;
     EFFECTS
     id = NEW(Pkt~id);
     'Current~ids() = Current~ids() UNION {id};

OFUN Set~tos(Type~of~serv tos; Pkt~id id);
     EXCEPTIONS
        Invalid~pkt(id);
        Invalid~tos(tos);

     EFFECTS
        'Type~of~service(id) = tos;

OFUN Set~ttl(INTEGER ttl; Pkt~id id);
     EXCEPTIONS
        Invalid~pkt(id);
        Too~small(ttl);
        RESOURCE~ERROR;
        $(ttl too large to fit in space allowed  by  lower  level
          representation)

     EFFECTS
        'Time~to~live(id) = ttl;

OFUN Set~protocol(Char~str pcol; Pkt~id id);
     EXCEPTIONS
        Invalid~pkt(id);
```

```
         Unknown~pcol(pcol);
      EFFECTS
         'Protocol(id) = pcol;

OFUN Set~dest~addr(Char~str da; Pkt~id id);
      EXCEPTIONS
         Invalid~pkt(id);
         Unknown~addr(da);
      EFFECTS
         'Dest~addr(id) = da;

OFUN Set~options(Optn opt; Pkt~id id);
      EXCEPTIONS
         Invalid~pkt(id);
         Unknown~opt(opt);
         RESOURCE~ERROR;
         $(This last exception is used to reflect a header
           overflow at the lower level without introducing the
           header size at this level.)

      EFFECTS
         'Option~field(id) = Option~field(id) UNION {opt};

OFUN Set~data(Data d; Pkt~id id);
      EXCEPTIONS
         RESOURCE~ERROR;
      EFFECTS
         Msg(id) = d;

OFUN Destroy(Pkt~id id);
      $(Allows a packet to be flushed from the system without
        sending.  It does not overwrite any information about the
        packet, it just makes the packet inaccessable since its id
        is no longer valid.  As part of the implementation the
        resources used by this packet could be reclaimed but the
        specification does not require this.)

      EXCEPTIONS
         Invalid~pkt(id);
      EFFECTS
         'Current~ids() = Current~ids() DIFF {id};

OFUN Dispatch(Pkt~id id);
      EXCEPTIONS
         Invalid~pkt(id);
      EFFECTS
         'Current~ids() = Current~ids() DIFF {id};
         EFFECTS~OF Send~to~net(Packets~in~prog(id));


END~MODULE
```

b. Module: Inet~recv

This module is essentially the complement of Inet~send. It allows an implementation to return all IP header fields at once or to get the packet identifier and use it to request only those fields needed as needed. This module has few exception conditions as most errors are not observable at the user interface. Packets arriving with an error in the header (i.e. a bad checksum) are not passed up to the user interface. At this level it is the same as if the network discarded that packet. Similarly, a misrouted packet never becomes available to the user.

```
MODULE Inet~recv

TYPES

     Byte: {0..2~Byte~size-1};
     Flag: BOOLEAN;
     Data: VECTOR~OF Byte;
     Char~str: VECTOR~OF CHAR;
     Optn: STRUCT~OF(Byte data, length; Data opt~info);
     Options: SET~OF Optn;
     Pkt~id: DESIGNATOR;
     Type~of~serv: STRUCT~OF(CHAR precedence; Flag strm;
                     CHAR reliability; Flag svsr, speed);
     Packet: STRUCT~OF(INTEGER ver; Type~of~serv tos; INTEGER ttl;
             Char~str pcol, sa, da; Options o; Data d);

PARAMETERS

     INTEGER Version;
     INTEGER Byte~size;
     INTEGER Max~pkt;

DEFINITI..

     BOOLEAN Invalid~pkt(Pkt~id id) IS NOT(id INSET Current~ids());
```

```
EXTERNALREFS

FROM Virt~net:
     OVFUN Recv~fm~net(Char~str da, pcol; BOOLEAN blk) -> Packet p;


FUNCTIONS

VFUN Type~of~service(Pkt~id id) -> Type~of~serv t;
     EXCEPTIONS
        Invalid~pkt(id);

     DERIVATION In~packets(id).tos;

VFUN Time~to~live(Pkt~id id) -> INTEGER t;
     HIDDEN;
     $(The value of this function is not available to the user
       interface; therefore, it is given as a hidden function.
       The module itself may wish to use this value for
       diagnostic and performance purposes but in some cases it
       does not have a defined value, the times when the packet
       was reassembled from fragments.)

     INITIALLY t = ?;

VFUN Protocol(Pkt~id id) -> Char~str p;
     EXCEPTIONS
        Invalid~pkt(id);

     DERIVATION In~packets(id).pcol;

VFUN Source~addr(Pkt~id id) -> Char~str s;
     EXCEPTIONS
        Invalid~pkt(id);

     DERIVATION In~packets(id).sa;

VFUN Dest~addr(Pkt~id id) -> Char~str d;
     EXCEPTIONS
        Invalid~pkt(id);

     DERIVATION In~packets(id).da;

VFUN Option~field(Pkt~id id) -> Options opt;
     EXCEPTIONS
        Invalid~pkt(id);

     DERIVATION In~packets(id).o;
```

```
VFUN Msg(Pkt~id id) -> Data m;
     EXCEPTIONS
        Invalid~pkt(id);

     DERIVATION In~packets(id).d;

VFUN Msg~length(Pkt~id id) -> INTEGER i;
     $(The interface description requires the IP to return the
       length of the buffer it passes back. This function makes
       that information available.)

     EXCEPTIONS
        Invalid~pkt(id);

     DERIVATION LENGTH(In~packets(id).d);

VFUN In~packets(Pkt~id id) -> Packet p;
     HIDDEN;
     INITIALLY p = ?;

VFUN Current~ids() -> SET~OF Pkt~id id;
     HIDDEN;
     INITIALLY id = { };

OVFUN Receive(Char~str da, pcol; BOOLEAN blk) -> Pkt~id id;
     $(The RESOURCE~ERROR exception can be mapped by a higher lev
       machine into either busy waiting or a software interrupt.)

     EXCEPTIONS
        RESOURCE~ERROR;

     EFFECTS
        id = NEW(Pkt~id);
        In~packets(id) = EFFECTS~OF Recv~fm~net(da, pcol, blk);
        In~packets(id).ttl > 0 =>
              'Current~ids() = Current~ids() UNION {id};

        $(The description states that packets are never delivered
          with time to live <= 0. The intent is to check this
          field whenever the header is processed so as to purge
          the net of undeliverable packets. This is an
          implementation decision; the above specifies only what
          will be true if the packet is delivered to the user
          interface.)


END~MODULE
```

c. Module: Virt~net

At the user interface the network can be viewed as a collection of packets bound for a particular protocol at a particular destination. The best data representation would be as a multiset(6). Unfortunately, Special does not support multiset as a primitive data type. It is possible to define a module in Special to implement multisets but it is a rather involved process (due to the lack of features to encapsulate data types) which would add nothing to this presentation. For this presentation we will assume that MULTISET~OF is a predefined type with operations to add and subtract elements.

It would be possible to use the VECTOR~OF construct to represent the network with the receive done on a random element. If properties about performance were of concern using a vector would probably present a construct closer to the properties that the actual implementation would have. The probability of receiving a packet could be based on its position in the queue which would be a function of when it was sent.

Some of the effects of the communication network on the actual data transfer can be modeled in a way similar to the suggestion of how to take into account parity errors in PSOS [Neu77]. This involves specifying OFUNs which have the error as their effects. This function can then be "called" by the offending "process". The verification

---

(6) A multiset is a structure similar to a set except that duplicates are allowed. Adding two multisets results in a multiset with each element occurring the sum of its occurrences in the original multisets. Subtraction behaves in a similar manner.

must take into account that the implementation behavior of an abstract machine is not known. In implementation proof it will still not be known except probabilistically. This means probabilistic methods will need to be introduced into the proof. The verification will give only a level of confidence.

```
MODULE Virt~net

TYPES

     Mset: MULTISET~OF Packet
     Char~str: VECTOR~OF CHAR;
     Options: SET~OF Char~str;
     Pkt~id: DESIGNATOR;
     Type~of~serv: STRUCT~OF(CHAR precedence; Flag strm;
                    CHAR reliability; Flag svsr, speed);
     Packet: STRUCT~OF(Type~of~serv tos; INTEGER ttl;
             Char~str pcol, sa, da; Options o; Data d);


FUNCTIONS

VFUN Network(Char~str da, pcol) -> Mset p;
     $(The network is the set of packets bound for a destination
       address. The packets are sorted by protocol at this level
       since a process receiving from the net will be doing the
       receive based on a call from a higher level protocol.)

     HIDDEN;
     INITIALLY p = ?;

OVFUN Recv~fm~net(Char~str da, pcol; BOOLEAN blk) -> Packet p;
     $(This function chooses a packet from those awaiting
       delivery, removes it from that set and returns it to the
       caller. This function points out another problem with
       Special. The description calls for receive to be either
       blocking or non-blocking. As shown here it is blocking,
       although a simple change could make it non-blocking. The
       problem is that Special does not allow this either/or.)

     DEFINITIONS
        Packet p IS p INSET Network(da, pcol);

     EXCEPTIONS
        RESOURCE~ERROR
```

```
     EFFECTS
        'Network(da, pcol) = Network(da, pcol) - {p};


OFUN Send~to~net(Packet p);
     $(This function places a packet into the set of packets for
       delivery to a particular destination address and
       protocol.)
     EXCEPTIONS
        RESOURCE~ERROR

     EFFECTS
        'Network(p.da, p.pcol) = Network(p.da, p.pcol) + {p};

OFUN Dup~pkt(Mset n);
     $(This function is "implemented" by the network. It
       duplicates an arbitrary packet.)
     DEFINITIONS
        Packet p IS p INSET n;

     EFFECTS
        'n = n + {p};


OFUN Drop~pkt(Mset n);
     $(This function is "implemented" by the network. It removes
       a packet from those to be delivered.)
     DEFINITIONS
        Packet p IS p INSET n;

     EFFECTS
        'n = n - {p};

END~MODULE
```

6. INTERNET PROTOCOL IN DETAIL

The above specification is only an interface specification. It would be adequate for a user of the IP interested only in sending datagrams; a user who did not need to use detailed knowledge about the implementation to improve efficiency. It is not an internal (implementation) specification. In operating system design an interface specification is often sufficient since two operating systems which present the same user interface will allow the same program

to run; the internal details are immaterial. In the realm of protocols, however, the internal specification is important as different implementations must be able to communicate (i. e. they must have "compatible" implementations).

Different networks may have different network access protocols; the implementation must provide for some means of moving datagrams between them. This is the responsibility of a network gateway. A gateway is a point where two networks in the catenet are interconnected. In the ARPA internet experiment a gateway [Str79] has three responsibilities: internet routing, fragmenting (but not reassembling) a datagram which is too large to pass through the net to which it is being routed and encapsulating and decapsulating internet datagrams with respect to the network access protocols of the nets being interconnected.

A gateway can be thought of as consisting of two halves - one at the entering point and one at the exit point. In this model each host contains half of a gateway. A gateway connecting more than two networks seems contradictory when drawn that way, since it contains more than two halves, but that is a function of the point of view - there is no contradiction.

The user interface specification deals completely with abstract objects (i.e. addresses are character strings). The detailed refinement has a series of abstract machines which provide the functionality to run interface specification on a network access protocol.

A trial decomposition is given in figure 7. The IP is decomposed into four layers: Host-Host virtual, Host-Host physical, Net-Net, and Net-Access. The first of these is the machine shown in the previous section. It provides an interface to those higher level protocols which require nothing more than a datagram service. The next lower abstract machine provides the necessary translations for the virtual datagram layer to be implemented. It can also be used as a visible interface by protocols which are interested in having some control over the transmission of its datagrams. The Net-Net layer is responsible for fragmentation and internet routing. The Net-Access layer is the abstract machine interface which the IP runs on.

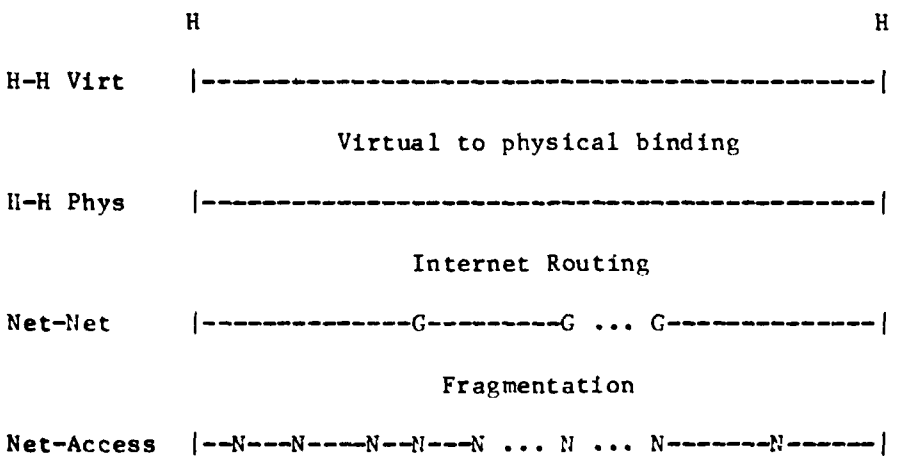


Fig. 7

This decomposition, though simple and straightforward, has a problem -- it requires that the Net-Net layer hide fragmentation from the hosts. To do this would require that datagrams be fragmented and reassembled on a network by network basis. The IP does not do this for two reasons. If gateways had to do reassembly all fragments of a

datagram would have to leave a net via the same gateway which would prevent load-sharing between gateways at the datagram level and requiring the gateways to do reassembly would require the gateway to be larger and more complex than not doing so. (It should be noted that the PUP architecture, using different design criteria, came up with a design which did require gateways to do reassembly. The above decomposition would probably be quite appropriate for the PUP architecture.)

Attempting to rearrange the levels to fit the IP led to the discovery that the initial guess fortuitiously placed the internet routing decision correctly with respect to fragmentation. It is not a free decision because, if a return route option (RRO) is present in the IP header, a datagram which did not need fragmenting before the RRO was updated can need fragmenting after the RRO is updated. After a few trials the decomposition in figure 8 was arrived at and used.

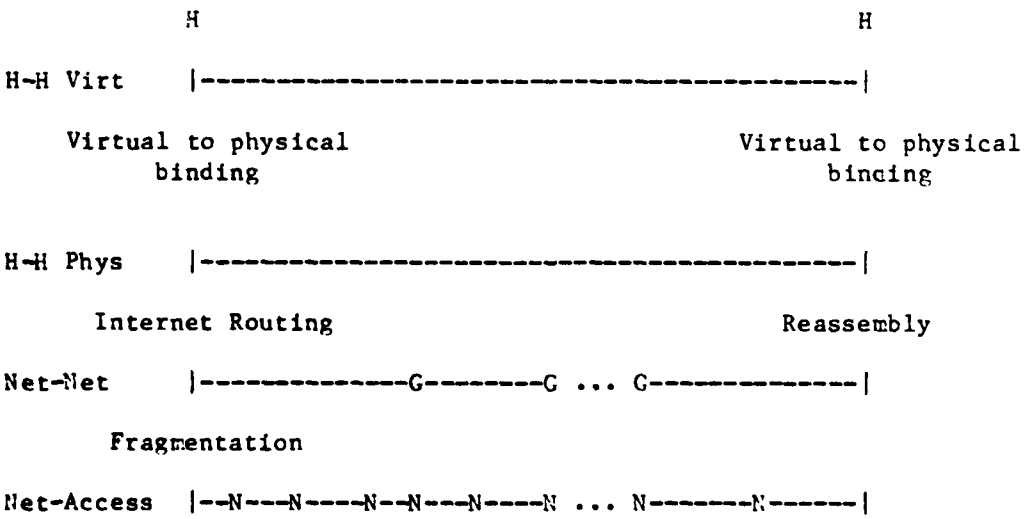


Fig. 8

The implementation was divided into ten modules distributed among four levels. The modules at each level are shown in figure 9.

Level 4:	Inet~send	Inet~recv	Virt~Net
Level 3:	Phys~send	Phys~recv	Phys~Net
Level 2:	Inet~rt	Reassemble	Phys~Net
Level 1:	Make~frag	Recv~frag	Phys~Net

Fig. 9

a. Module: Phys~send

This module provides physical representation for the virtual objects defined in the Inet~send module. It also includes the IP header fields omitted from the Inet~send.

```
MODULE Phys~send

TYPES

     Quartet: {0..15};
     Byte: {0..2~Byte~size-1};
     Net~id: Byte;
     Phys~addr: {VECTOR~OF Byte p | LENGTH(p) = Addr~size};
     Inet~addr: STRUCT~OF(Net~id net; Phys~addr a);
     Half~wd: {0..2^(Word~size/2)-1};
     Flag: BOOLEAN;
     Data: VECTOR~OF Byte;
     Char~str: VECTOR~OF CHAR;
     Pkt~id: DESIGNATOR;
     Type~of~serv: STRUCT~OF(CHAR precedence; Flag strm;
                     CHAR reliability; Flag svsr, speed);
     Packet: STRUCT~OF(Quartet v; Type~of~serv tos; Byte ttl, pcol;
             Inet~addr sa, da; Data o, d);


PARAMETERS

     Quartet Version;
     SET~OF Net~id Known~nets;
     SET~OF Net~id Neighbors;
     SET~OF Inet~addr Source~addrs;
```

```
    Inet~addr Default~source~addr;
    SET~OF Byte Non~dup~optn;

DEFINITIONS

    BOOLEAN Invalid~net(Inet~addr da) IS
                          NOT(da.net INSET Known~nets);
    $(The IP is not responsible for interpreting network
      specific addresses. All it can check is that the network
      field addresses a network it can knows about.)

    BOOLEAN Invalid~sa(Inet~addr sa) IS NOT(sa INSET Source~addrs);

    BOOLEAN Invalid~pkt(Pkt~id id) IS NOT(id INSET Current~ids());


EXTERNALREFS

    FROM Phys~net:
       OFUN Send~to~net(Packet p);
      INTEGER Byte~size;                $(Currently 8 bits)
      INTEGER Addr~size;                $(Currently 3 bytes)
      INTEGER Word~size;                $(Currently 4 bytes)

    FROM Inet~rt:
      INTEGER Max~inet~pkt;          $(Currently 2**16-1 bytes)
      INTEGER Max~inet~hdr;          $(Currently 15 words, 60 bytes)
      INTEGER Min~inet~hdr;          $(Currently 5 words, 20 bytes)

ASSERTIONS

    Default~source~addr INSET Source~addrs;

    $(This assertion bounds the size of the IP header)
    FORALL Pkt~id id:
    LENGTH(Option~field(id)) <= Max~inet~hdr - Min~inet~hdr;

    $(This assertion bounds the size of an IP datagram.)
    FORALL Pkt~id id:
    Min~inet~hdr + LENGTH(Option~field(id)) + LENGTH(Msg(id))
    <= Max~inet~pkt;

FUNCTIONS

VFUN Type~of~service(Pkt~id id) -> Type~of~serv tos;
    HIDDEN;
    INITIALLY tos = ?;

VFUN Time~to~live(Pkt~id id) -> INTEGER ttl;
    $(Time measured in seconds)
    HIDDEN;
    INITIALLY ttl = ?;
```

```
VFUN Protocol(Pkt~id id) -> Quartet p;
     HIDDEN;
     INITIALLY pcol = ?;

VFUN Source~addr(Pkt~id id) -> Inet~addr s;
     HIDDEN;
     INITIALLY sa = Default~source~addr;

VFUN Dest~addr(Pkt~id id) -> Inet~addr d;
     HIDDEN;
     INITIALLY da = ?;

VFUN Option~field(Pkt~id id) -> Data opt;
     HIDDEN;
     INITIALLY o = ?;

VFUN Msg(Pkt~id id) -> Data m;
     HIDDEN;
     INITIALLY m = ?;

VFUN Packets~in~prog(Pkt~id id) -> Packet p;
     EXCEPTIONS
        Invalid~pkt(id);

     DERIVATION
        STRUCT (Version,
        Type~of~service(id),
        Time~to~live(id),
        Protocol(id),
        Source~addr(id),
        Dest~addr(id),
        Option~field(id),
        Msg(id));

VFUN Current~ids() -> SET~OF Pkt~id id;
     HIDDEN;
     INITIALLY id = { };

OVFUN Make~pkt() -> Pkt~id id;
     EFFECTS
        id = NEW(Pkt~id);
        'Current~ids() = Current~ids() UNION {id};

OFUN Set~tos(Type~of~serv tos; Pkt~id id);
     $(The description places no restriction on setting the fields
       of the type of service.  In the future it may be necessary
       to add some checking on the use of the priority field.)
     EXCEPTIONS
        Invalid~pkt(id);

     EFFECTS
        'Type~of~service(id) = tos;
```

```
OFUN Set~ttl(Byte ttl; Pkt~id id);
     $(Time measured in seconds)
     EXCEPTIONS
        Invalid~pkt(id);

     EFFECTS
        'Time~to~live(id) = ttl;

OFUN Set~protocol(Quartet pcol; Pkt~id id);
     EXCEPTIONS
        Invalid~pkt(id);

     EFFECTS
        'Protocol(id) = pcol;

OFUN Set~source~addr(Inet~addr sa; Pkt~id id);
     EXCEPTIONS
        Invalid~pkt(id);
        Invalid~sa(sa);

     EFFECTS
        'Source~addr(id) = sa;

OFUN Set~dest~addr(Inet~addr da; Pkt~id id);
     EXCEPTIONS
        Invalid~pkt(id);
        Invalid~net(da);

     EFFECTS
        'Dest~addr(id) = da;

OFUN Set~options(Data opt; Pkt~id id);
     EXCEPTIONS
        Invalid~pkt(id);

     EFFECTS
        'Option~field(id) = opt;

OFUN Set~Msg(Data m; Pkt~id id);
     EXCEPTIONS
        Invalid~pkt(id);

     EFFECTS
        'Msg(id) = m;

OFUN Destroy(Pkt~id id);
     $(See comment in Inet~send module)
     EXCEPTIONS
        Invalid~pkt(id);

     EFFECTS
        'Current~ids() = Current~ids() DIFF {id};
```

```
OFUN Dispatch(Pkt~id id);
     EXCEPTIONS
        Invalid~pkt(id);

     EFFECTS
        'Current~ids() = Current~ids() DIFF {id};
        EFFECTS~OF Send~to~net(Packets~in~prop(id));


END~MODULE
```

b. Module: 'Phys~recv

This module is the counterpart of the Phys~send module on the receive side.

```
MODULE Phys~recv

TYPES

     Quartet: {0..15};
     Byte: {0..2^Byte~size-1};
     Net~id: Byte;
     Phys~addr: {VECTOR~OF Byte p | LENGTH(p) = Addr~size};
     Inet~addr: STRUCT~OF(Net~id net; Phys~addr a);
     Half~wd: {0..2^(Word~size/2)-1};
     Flag: BOOLEAN;
     Data: VECTOR~OF Byte;
     Char~str: VECTOR~OF CHAR;
     Pkt~id: DESIGNATOR;
     Type~of~serv: STRUCT~OF(CHAR precedence; Flag strm;
                     CHAR reliability; Flag svsr, speed);
     Packet: STRUCT~OF(Quartet v; Type~of~serv tos; Byte ttl, pcol;
             Inet~addr sa, da; Data o, d);


PARAMETERS

     Quartet Version;
     SET~OF Inet~addr Dest~addrs;

DEFINITIONS

     BOOLEAN Invalid~pkt(Pkt~id id) IS NOT(id INSET Current~ids());

     BOOLEAN Invalid~da(Inet~addr da) IS NOT(da INSET Dest~addrs);
```

```
EXTERNALREFS

     FROM Phys~net:
        OVFUN Remove~fm~net(Inet~addr da) -> Packet p;
        INTEGER Byte~size;                  $(Currently 8 bits)
        INTEGER Addr~size;                  $(Currently 3 bytes)
        INTEGER Word~size;                  $(Currently 4 bytes)

     FROM Inet~rt:
       INTEGER Max~inet~pkt;          $(Currently 2**16-1 bytes)
       INTEGER Max~inet~hdr;          $(Currently 15 words, 60 bytes)
       INTEGER Min~inet~hdr;          $(Currently 5 words, 20 bytes)


ASSERTIONS

     FORALL Pkt~id id:
     LENGTH(Option~field(id)) <= Max~inet~hdr - Min~inet~hdr;

     FORALL Pkt~id id:
     Min~inet~hdr + LENGTH(Option~field(id)) + LENGTH(Msg(id))
     <= Max~inet~pkt;


FUNCTIONS

VFUN Type~of~service(Pkt~id id) -> Type~of~serv t;
     EXCEPTIONS
        Invalid~pkt(id);

     DERIVATION In~packets(id).tos;

VFUN Time~to~live(Pkt~id id) -> INTEGER t;
     EXCEPTIONS
        Invalid~pkt(id);

     DERIVATION In~packets(id).ttl;

VFUN Protocol(Pkt~id id) -> Quartet p;
     EXCEPTIONS
        Invalid~pkt(id);

     DERIVATION In~packets(id).pcol;

VFUN Source~addr(Pkt~id id) -> Inet~addr s;
     EXCEPTIONS
        Invalid~pkt(id);

     DERIVATION In~packets(id).sa;
```

```
VFUN Dest~addr(Pkt~id id) -> Inet~addr d;
     EXCEPTIONS
        Invalid~pkt(id);

     DERIVATION In~packets(id).da;

VFUN Option~field(Pkt~id id) -> Data opt;
     EXCEPTIONS
        Invalid~pkt(id);

     DERIVATION In~packets(id).o;

VFUN Msg(Pkt~id id) -> Data m;
     EXCEPTIONS
        Invalid~pkt(id);

     DERIVATION In~packets(id).d;

VFUN In~packets(Pkt~id id) -> Packet p;
     HIDDEN;

     INITIALLY p = ?;

VFUN Current~ids() -> SET~OF Pkt~id id;
     HIDDEN;
     INITIALLY id = { };

OVFUN Recv~pkt(Inet~addr da) -> Pkt~id id;
     EXCEPTIONS
        Invalid~da(da);
        RESOURCE~ERROR;

     EFFECTS
        id = NEW(Pkt~id);
        'Current~ids() =  {id} UNION Current~ids();
        'In~packets(id) = EFFECTS~OF Remove~fm~net(da);


END~MODULE
```

c. Module: Phys~net

At this level the network is viewed as a collection of packets bound for a destination. The packets are not segregated by the particular higher level protocol to which they are directed.

```
MODULE Phys~net

TYPES

     Byte: {0..2~Byte~size-1};
     Data: VECTOR~OF Byte;
     Net~id: Byte;
     Phys~addr: {VECTOR~OF Byte p | LENGTH(p) = Addr~size};
     Inet~addr: STRUCT~OF(Net~id net; Phys~addr a);
     Half~wd: {0..2~(Word~size/2)-1};
     Mset: MULTISET~OF Packet
     Pkt~id: DESIGNATOR;
     Type~of~serv: STRUCT~OF(CHAR precedence; Flag strm;
                     CHAR reliability; Flag svsr, speed);
     Inet~hdr: STRUCT~OF(Version v; Quartet ihl; Type~of~serv tos;
             Half~wd tot~ln, indent; Flag df, mf; Byte ttl, pcol;
             Inet~addr sa, da; Data opt);
     Packet: STRUCT~OF(Inet~hdr ih; Data msg; BOOLEAN check~sum);


PARAMETERS

     INTEGER Byte~size;                $(Currently 8 bits)
     INTEGER Addr~size;                $(Currently 3 bytes)
     INTEGER Word~size;                $(Currently 4 bytes)


FUNCTIONS

VFUN Network(Inet~addr da) -> Mset p;
     $(The network is the set of packets bound for a destination
       address.)
     HIDDEN;
     INITIALLY p = { };

OVFUN Remove~rm~net(Inet~addr da) -> Packet p;
     LET p INSET Network(da);
     EXCEPTIONS
        Empty(da);

     EFFECTS
        'Network(da) = Network(da) - {p};

OFUN Send~to~net(Packet p);
     EXCEPTIONS
        Full(p.da);

     EFFECTS
        'Network(p.da) = Network(p.da) + {p};

OFUN Dup~pkt(Addr da);
     LET p INSET Network(da);
```

```
     EFFECTS
        'Network(da) = Network(da) + {p};


OFUN Drop~pkt(Inet~addr da);
     LET p INSET Network(da);

     EFFECTS
        'Network(da) = Network(da) - {p};


OFUN Modify~pkt(Packet p);
     $(Since the IP only checksums the internet header modifying
       the data has no visible effect as far as IP is concerned.
       The second clause of the EFFECTS paragraph is there since
       Special explicitly declares that all values not mentioned
       in an EFFECTS section are not modified.)

     EFFECTS
        'p.ih ~= p.ih => 'p.check~sum = FALSE;
        'p.msg = p.msg OR 'p.msg ~= p.msg;


END~MODULE
```

d. Module: Inet~rt

This module contains the functions to update the routing options of the IP. Since the RPO is in terms of the net the datagram is entering it must have the necessary functions to decide which net this will be. This module also handles the other options of the header.

```
MODULE Inet~rt

TYPES

     Quartet: {0..15};
     Byte: {0..2~Byte~size-1};
     Flag: BOOLEAN;
     Net~id: Byte;
     Data: VECTOR~OF Byte;
     Phys~addr: {VECTOR~OF Byte p | LENGTH(p) = Addr~size};
     Inet~addr: STRUCT~OF(Net~id net; Phys~addr a);
     Half~wd: {0..2~(Word~size/2)-1};
     Pkt~id: DESIGNATOR;
```

```
Type~of~serv: STRUCT~OF(CHAR precedence; Flag strm;
                  CHAR reliability; Flag svsr, speed);
Route: {VECTOR~OF Inet~addr r | LENGTH(r) < 10};
Route~opt: STRUCT~OF(Byte opt~type, length; Route rt);
Sec~opt: STRUCT~OF(Byte opt~type, length, sec~lvl, tcc);
Stream: {VECTOR~OF Byte s | LENGTH(s) = Stream~id~ln};
Stream~opt: STRUCT~OF(Byte opt~type, length; Stream s);
Timest: {VECTOR~OF Byte t | LENGTH(t) = Timest~ln};
Timest~opt: STRUCT~OF(Byte opt~type, length; Timest t);

$(It is unclear how to handle multiple source route, return
  route or security options. The protocol description
  neither prohibits such packets nor tells what action to
  take if one arrives. The specifications do not allow such
  packets.)

Packet: STRUCT~OF(Quartet version, ihl; Type~of~serv tos;
        Half~wd totln, indent; Flag df, mf; Byte ttl, pcol;
        Inet~addr sa, da; Route~opt rro, sro; Sec~opt sec;
        SET~OF Stream~opt str; SET~OF Timest~opt time; Data d);
```

PARAMETERS

```
Inet~addr Net~addr;                  $(Address for RRO)
INTEGER Max~inet~pkt;                $(Currently 2**16-1 Bytes)
INTEGER Max~inet~hdr;                $(Currently 60 Bytes)
INTEGER Min~inet~hdr;                $(Currently 20 Bytes)
INTEGER Stream~id~ln;                $(Currently 2)
INTEGER Timest~ln;                   $(Currently 4)
SET~OF Byte Known~nets;
SET~OF Byte Valid~timest~opts;       $(Currently Satnet and
                                       Internet)
Byte Source~route;
Byte Return~route;
```

DEFINITIONS

```
BOOLEAN Unknown~net(Net~id net) IS NOT (net INSET Known~nets);

BOOLEAN No~rr~opt(Packet p) IS p.rro = ?;

BOOLEAN Empty~sr(Route~opt r) IS LENGTH(r.rt) = 0;

BOOLEAN Not~empty(Route~opt r) IS LENGTH(r.rt) ~= 0;

BOOLEAN Not~sro(Route~opt r) IS
                          NOT (r.opt~type = Source~route);

BOOLEAN Not~rro(Route~opt r) IS
                          NOT (r.opt~type = Return~route);
```

```
     BOOLEAN Invalid~ts~opt(Timest~opt t) IS NOT(t.opt~type INSET
             Valid~timest~opts);

     BOOLEAN Invalid~pkt(Pkt~id id) IS NOT(id INSET Current~ids());


EXTERNALREFS

     FROM Phys~net:
        INTEGER  Byte~size;                  $(Currently 8)
        INTEGER  Addr~size;                  $(Currently 3)
        INTEGER  Word~size;                  $(Currently 4)


FUNCTIONS

VFUN Packets(Pkt~id id) -> Packet p;
     EXCEPTIONS
        Invalid~pkt(id);

     INITIALLY p = ?;

VFUN Current~ids() -> SET~OF Pkt~id id;
     HIDDEN;
     INITIALLY id = { };

VFUN Next~dest(Pkt~id id) -> Inet~addr d;
     DEFINITIONS
        Route~opt r IS Packets(id).sro;

     EXCEPTIONS
        Invalid~pkt(id);
        Empty~sr(Packets(id).sro);

     DERIVATION r.rt[(r.length -2)/Word~size];

OFUN Update~sr(Pkt~id id);
     EXCEPTIONS
        Invalid~pkt(id);
        Empty~sr(Packets(id).sro);

     EFFECTS
        'Packets(id).da = Next~dest(id);
        'Packets(id).sro.length = Packets(id).sro.length - Word~size;
        'Packets(id).ihl = Packets(id).ihl - 1;
        'Packets(id).totln = Packets(id).totln - Word~size;

     $(Note: the header length is measured in words whereas the
       total length is in octets.)
```

```
OFUN Update~rr(Pkt~id id);
     EXCEPTIONS
        Invalid~pkt(id);
        No~rr~opt(Packets(id));
        RESOURCE~ERROR;

     EFFECTS
        'Packets(id).totln = Packets(id).totln + Word~size;
        'Packets(id).ihl = Packets(id).ihl + 1;
        'Packets(id).rro.lenght = Packets(id).rro.length + Word~size;
        'Packets(id).rro.rt[(Packets(id).rro.length -2)/Word~size]
                                                    = Net~addr;

OFUN Set~sro(Pkt~id id; Route~opt sr);
     EXCEPTIONS
        Invalid~pkt(id);
        Not~sro(sr);

     EFFECTS
        'Packets(id).sro = sr;

OFUN Request~rro(Pkt~id id; Route~opt rr);
     EXCEPTIONS
        Invalid~pkt(id);
        Not~rro(rr);
        Not~empty(rr);
        RESOURCE~ERROR;

     EFFECTS
        'Packets(id).rro = rr;

OFUN Set~sec~opt(Pkt~id id; Sec~opt s);
     EXCEPTIONS
        Invalid~pkt(id);
        RESOURCE~ERROR;

     EFFECTS
        'Packets(id).sec = s;

OFUN Set~str~opt(Pkt~id id; Stream~opt s);
     EXCEPTIONS
        Invalid~pkt(id);
        RESOURCE~ERROR;

     EFFECTS
        'Packets(id).str = Packets(id).str UNION {s};

OFUN Set~time~opt(Pkt~id id; Timest~opt t);
     EXCEPTIONS
        Invalid~pkt(id);
        Invalid~ts~opt(t);
        RESOURCE~ERROR;
```

```
     EFFECTS
        'Packets(id).time = Packets(id).time UNION {t};

$(The next two functions provide the interface level of the
  gateway to gateway protocol [Str79]. They are responsible for
  maintaining and updating routing information for the hosts
  within the internet.)

VFUN Next~net(Net~id dn; Type~of~serv tos) -> Net~id n;
     HIDDEN;
     INITIALLY n = ?;

OFUN Update~rt~info(Net~id dest~net, nxt~net; Type~of~serv tos);
     EXCEPTIONS
        Unknown~net(dest~net);
        Unknown~net(nxt~net);

     EFFECTS
        'Next~net(dest~net, tos) = nxt~net;

END~MODULE
```

e. Module: Reassemble

This module maps datagram fragments into complete datagrams. Its purpose is to present only "whole" datagrams to the Recv~phys level.

```
MODULE Reassemble

TYPES

     Quartet: {0..15};
     Byte: {0..2~Byte~size-1};
     Offset: {0..Max~offset};
     Net~id: Byte;
     Phys~addr: {VECTOR~OF Byte p | LENGTH(p) = Addr~size};
     Inet~addr: STRUCT~OF(Net~id net; Phys~addr a);
     Flag: BOOLEAN;
     Half~wd: {0..2~(Word~size/2)-1};
     Data: VECTOR~OF Byte;
     Type~of~serv: STRUCT~OF(CHAR precedence; Flag strm;
                     CHAR reliability; Flag svsr, speed);
     Packet: STRUCT~OF(Quartet v, ihl; Type~of~serv tos;
             Half~wd totln, ident; Flag df, mf; Offset fo;
             Byte ttl, pcol; Inet~addr sa, da; Data opt, d);
```

```
PARAMETERS

     Quartet Version;
     INTEGER Max~offset;
     SET~OF Byte Known~pcols;
     SET~OF Byte Known~nets;


DEFINITIONS

     BOOLEAN Incomplete~datagram(Half~wd id; Offset f; Byte pcol;
         Inet~addr sa, da) IS NOT Pkt~complete(id, pcol, sa, da);

     BOOLEAN Bad~pcol(Packet p) IS NOT (p.pcol INSET Known~pcols);

     BOOLEAN Not~frag(Packet p) IS p.mf = FALSE AND p.fo = 0;

     BOOLEAN No~last~frag(Half~wd id; Byte pcol; Inet~addr sa, da)
        IS NOT (EXISTS Offset f: More~frags(id, f, pcol, sa, da) =
                TRUE);

EXTERNALREFS

     FROM Phys~net:
        INTEGER Byte~size;                  $(Currently 8)
        INTEGER Addr~size;                  $(Currently 3)
        INTEGER Word~size;                  $(Currently 4)

     FROM Inet~rt:
        INTEGER Min~inet~hdr;           $(Currently 20 Bytes)


FUNCTIONS

VFUN Header~ln(Half~wd id; Offset f; Byte pcol; Inet~addr sa, da)
                                                -> Quartet ihl;
     HIDDEN;
     DEFINITIONS
        INTEGER l IS LENGTH(Options(id, f, pcol, sa, da));

     DERIVATION Min~inet~hdr + l/Word~size;

VFUN Total~ln(Half~wd id; Offset    byte pcol; Inet~addr sa, da)
                                              -> INTEGER i;
     HIDDEN;
     INITIALLY i = 0;

$(The following four VFUNs have no counterpart in the reassembled
  datagram. The description gives no method to derive them from
  the fragments of a datagram nor are they returned at the
  interface to the IP.)
```

```
VFUN Ident(Half~wd id; Offset f; Byte pcol; Inet~addr sa, da)
                                          -> INTEGER i;
     HIDDEN;
     INITIALLY i = 0;

VFUN More~frags(Half~wd id; Offset f; Byte pcol; Inet~addr sa, da)
                                                -> BOOLEAN m;
     HIDDEN;
     INITIALLY m = ?;

VFUN Frag~offset(Half~wd id; Offset f; Byte pcol; Inet~addr sa, da)
                                                -> INTEGER i;
     HIDDEN;
     INITIALLY i = 0;

VFUN Time~to~live(Half~wd id; Offset f; Byte pcol; Inet~addr sa, da)
                                                -> INTEGER t;
     HIDDEN;
     INITIALLY t = 0;

VFUN Options(Half~wd id; Offset f; Byte pcol; Inet~addr sa, da)
                                                -> Data o;
     HIDDEN;
     INITIALLY o = ?;

VFUN Msg(Half~wd id; Offset f; Byte pcol; Inet~addr sa, da)
                                                -> Data m;
     HIDDEN;
     INITIALLY m = ?;

VFUN Checksum(Half~wd id; Offset f; Byte pcol; Inet~addr sa, da)
                                                -> INTEGER c;
     HIDDEN;
     DERIVATION ?;

VFUN Frag~blk~table(Half~wd id; Byte pcol; Inet~addr sa, da;
                    INTEGER blk~no) -> BOOLEAN m;
     $(The table keeps track of which pieces of a datagram
       have been received)
     HIDDEN;
     INITIALLY m = ?;

VFUN Pkt~complete(Half~wd id; Byte pcol; Inet~addr sa, da)
                                                -> BOOLEAN m;
     DEFINITIONS
        INTEGER Max~blk IS LET INTEGER f |
                More~frags(id, f, pcol, sa, da) = FALSE
                IN Frag~offset(id, f, pcol, sa, da) +
                Total~ln(id, f, pcol, sa, da) -
                Header~ln(id, f, pcol, sa, da);
```

```
     EXCEPTIONS
        No~last~frag(id, pcol, sa, da);

     DERIVATION FORALL INTEGER i | 0 < i AND i <= Max~blk:
                Frag~blk~table(id, pcol, sa, da, i) = TRUE;

$(The following functions present the visible interface of this
  module. They make available the fields of the datagram after
  reassembly. The R preceeding the name indicates that it is for
  a reassembled datagram.)

VFUN R~hdr~ln(Half~wd id; Byte pcol; Inet~addr sa, da)
                                                  -> INTEGER ln;
     DEFINITIONS
        INTEGER l IS LENGTH(R~options(id, pcol, sa, da));

     EXCEPTIONS
        Incomplete~datagram(id, pcol, sa, da);

     DERIVATION Min~inet~hdr + l/Word~size;

VFUN R~type~of~service(Half~wd id; Byte pcol; Inet~addr sa, da) ->
                                                 Type~of~serv tos;
     EXCEPTIONS
        Incomplete~datagram(id, pcol, sa, da);

     INITIALLY tos = ?;

VFUN R~total~ln(Half~wd id; Byte pcol; Inet~addr sa, da) -> INTEGER l;
     EXCEPTIONS
        Incomplete~datagram(id, pcol, sa, da);

     INITIALLY l = ?;

VFUN R~options(Half~wd id; Byte pcol; Inet~addr sa, da) -> Data o;
     EXCEPTIONS
        Incomplete~datagram(id, pcol, sa, da);

     INITIALLY o = ?;

VFUN R~data(Half~wd id; Byte pcol; Inet~addr sa, da) -> Data m;
     EXCEPTIONS
        Incomplete~datagram(id, pcol, sa, da);

     INITIALLY m = ?;

OFUN Reas~frag(Packet p);
     DEFINITIONS
        INTEGER st IS 8*p.fo;
        INTEGER fn IS st + p.totln - p.ihl;
```

```
EXCEPTIONS
   Not~frag(p);
   Bad~pcol(p);

EFFECTS
   'R~type~of~service(p.ident, p.pcol, p.sa, p.da) =  p.tos;
   p.fo = 0 =>
        'R~options(p.ident, p.pcol, p.sa, p.da) = p.opt;
   FORALL INTEGER i| st <= i AND i <= fn:
   'R~data(p.ident, p.pcol, p.sa, p.da)[i] = p.d[i-st];
   FORALL INTEGER i| p.fo < i AND i < (p.totln/8 - p.fo):
      'Frag~blk~table(p.ident, p.pcol, p.sa, p.da, i) = TRUE;

   $(The total lenght must be divided by eight  because  the
     fragment offset is measured in units of eight bytes.)

END~MODULE
```

f. Module: Send~frag

This module implements fragmentation at the source end. This module provides a method of fragmenting datagrams so that unique reassembly can take place. Datagrams are identified by four items: the protocol above the IP which is sending them, the source address, the destination address and the identification field. Any two datagrams with these fields identical are considered identical by the IP. To distinguish between fragments of the same datagram the fragment offset and total length field (or more fragments field) are needed. Since this would require the next level to keep track of offsets, unique identifiers are generated. The next level can then simply ask for the identifiers which correspond to a given datagram.

```
MODULE Send~frag

TYPES

     Quartet: {0..15};
     Byte: {0..2^Byte~size-1};
     Half~wd: {0..2^(Word~size/2)-1};
```

```
    Offset: {0..Max~offset};
    Net~id: Byte;
    Flag: BOOLEAN;
    Frag~des: DESIGNATOR;
    Local~addr: {VECTOR~OF Byte a | LENGTH(a) = Addr~size};
    Inet~addr: STRUCT~OF(Net~id net; Local~addr la);
    Data: VECTOR~OF Byte;
    Type~of~serv: STRUCT~OF(CHAR precedence; Flag strm;
                    CHAR reliability; Flag svsr, speed);
    Packet: STRUCT~OF(Quartet v, ihl; Type~of~serv tos;
            Half~wd totln, ident; Flag df, mf; Offset fo;
            Byte ttl, pcol; Inet~addr sa, da; Data opt, d);

    $(Although the order of fields in a structure is irrevelent
      as far as SPECIAL is concerned they are ordered here as
      actually implemented to make the mapping to the implementation
      as straight forward as possible.)

PARAMETERS

    Quartet Version;
    INTEGER Max~frag;
    INTEGER Max~offset;
    SET~OF Byte Known~pcols;
    SET~OF Byte Supplying~ids;
    SET~OF Net~id Known~nets;
    SET~OF Net~id Neighbors;


DEFINITIONS

    BOOLEAN Unknown~net(Net~id n) IS NOT(n INSET Known~nets);

    BOOLEAN Poor~pcol(Byte p) IS NOT(p INSET Supplying~ids);

    BOOLEAN Df~set(Packet p) IS p.df = TRUE;

    BOOLEAN Too~big(INTEGER p) IS p > Max~frag;

    BOOLEAN Bad~start~pos(INTEGER i, j) IS (i MOD Byte~size ~=0)
                          OR (i >= j);

    BOOLEAN Bad~end~pos(Packet p; INTEGER i) IS (i MOD Byte~size ~=0)
                        AND (p.d[i+1] ~= ?);
    $(All fragments except the last one must be a multiple of
      eight in length.)


EXTERNALREFS

    FROM Phys~net:
       OFUN Send~to~net(Packet p);
```

```
      INTEGER Byte~size;                $(Currently 8)
      INTEGER Addr~size;                $(Currently 3)
      INTEGER Word~size;                $(Currently 4)


   FROM Inet~rt:
      INTEGER Max~inet~hdr;          $(Currently 60 bytes)
      INTEGER Min~inet~hdr;          $(Currently 20 bytes)

ASSERTIONS

   FORALL Packet p: LENGTH(p.opt) MOD Word~size = 0;

   FORALL Packet p: LENGTH(p.opt) <= Max~inet~hdr - Min~inet~hdr;

   FORALL Packet p: p.df = TRUE => p.mf = FALSE AND p.fo = 0;

   $(This is the intent of the current description although it
     is not explicitly stated as such)


FUNCTIONS

VFUN F~Header~ln(Frag~des id) -> Quartet ihl;
   HIDDEN;
   DEFINITIONS
      INTEGER l IS LENGTH(F~Options(id));

   DERIVATION Min~inet~hdr + l/Word~size;

VFUN F~Type~of~service(Frag~des id) -> Type~of~serv t;
   HIDDEN;
   INITIALLY t = ?;

VFUN F~Total~ln(Frag~des id) -> Half~wd i;
   HIDDEN;
   INITIALLY i = 0;

VFUN F~Ident(Frag~des id) -> Half~wd i;
   HIDDEN;
   INITIALLY i = 0;

VFUN F~Dont~frag(Frag~des id) -> BOOLEAN df;
   HIDDEN;
   INITIALLY df = FALSE;

   VFUN F~More~frags(Frag~des id) -> BOOLEAN mf;

      INITIALLY mf = FALSE;
```

```
VFUN F~Frag~offset(Frag~des id) -> Half~wd i;
     HIDDEN;
     INITIALLY i = 0;

VFUN F~Time~to~live(Frag~des id) -> Byte t;
     HIDDEN;
     INITIALLY t = 0;

VFUN F~Protocol(Frag~des id) -> Byte p;
     HIDDEN;
     INITIALLY p = 0;

VFUN F~Source~addr(Frag~des id) -> Inet~addr s;
     HIDDEN;
     INITIALLY s = ?;

VFUN F~Dest~addr(Frag~des id) -> Inet~addr d;
     HIDDEN;
     INITIALLY d = ?;

VFUN F~Options(Frag~des id) -> Data o;
     HIDDEN;
     INITIALLY o = ?;

VFUN F~Msg(Frag~des id) -> Data m;
     HIDDEN;
     INITIALLY m = ?;

VFUN F~Checksum(Frag~des id) -> Half~wd c;
     HIDDEN;
     DERIVATION ?;

VFUN Fragments(Frag~des id) -> Packet p;
     HIDDEN;
     DERIVATION
        STRUCT (Version,
        F~Header~ln(id),
        F~Type~of~service(id),
        F~Total~ln(id),
        F~Ident(id),
        F~Dont~frag(id),
        F~More~frags(id),
        F~Frag~offset(id),
        F~Time~to~live(id),
        F~Protocol(id),
        F~Source~addr(id),
        F~Dest~addr(id),
        F~Options(id),
        F~Msg(id),
        F~Checksum(id));
```

```
OFUN Dispatch(Frag~des id);
     EXCEPTIONS
        RESOURCE~ERROR;

     EFFECTS
        EFFECTS~OF Send~to~net(Fragments(id));

VFUN Frags~of~pkt(Half~wd id; Byte pcol; Inet~addr sa, da)
                                          -> SET~OF Frag~des fd;
     EXCEPTIONS
        Poor~pcol(pcol);
        Unknown~net(sa.net);
        Unknown~net(da.net);

     INITIALLY fd = { };

OVFUN Fragment(Packet p; INTEGER srt~pos, end~pos) -> Frag~des id;
     $(In the IP description the fragmentation algorithm is given
       as an operational definition.  This specification embodies
       the same concept as an abstract machine definition.)
     EXCEPTIONS
        Df~set(p.ident, p.pcol, p.sa, p.da);
        Bad~start~pos(srt~pos, end~pos);
        Bad~end~pos(p, end~pos);
        Too~big(end~pos - srt~pos);

     EFFECTS
        id = NEW(Frag~des);
        'Frags~of~pkt(p.ident, p.pcol, p.sa, p.da) =
                Frags~of~pkt(p.ident, p.pcol, p.sa, p.da) UNION {id};
        'F~Type~of~service(id) = p.tos;
        'F~Ident(id) = p.ident;
        $(The don't fragment flag need not be set as the initial value
          is correct.  The same is true for the more fragments flag
          for be last fragment.)
        end~pos ~= LENGTH(p.d) => 'F~More~frags(id) = TRUE;
        'F~Frag~offset(id) = srt~pos/(2*Word~size);
        'F~Time~to~live(id) = p.ttl;
        'F~Protocol(id) = p.pcol;
        'F~Source~addr(id) = p.sa;
        'F~Dest~addr(id) = p.da;
        srt~pos = 0 => 'F~Options(id) =
                First~frag~opts(p.ident, p.pcol, p.sa, p.da);
        srt~pos ~= 0 => 'F~Options(id) =
                All~frag~opts(p.ident, p.pcol, p.sa, p.da);
        'F~Msg(id) = VECTOR(FOR i FROM srt~pos TO end~pos: p.d[i]);

END~MODULE
```

g. Module: Recv~frag

Although this module is the counterpart of the module Send~frag, it is much simpler because it is not responsible for reassembling the fragments created by Send~frag. Because datagrams are not reassembled at this level, the problem of distinguishing between a datagram and its first fragment does not reoccur here.

```
MODULE Recv~frag

TYPES

     Quartet: {0..15};
     Byte: {0..2^Byte~size-1};
     Half~wd: {0..2^(Word~size/2)-1};
     Net~id: Byte;
     Offset: {0..Max~offset};
     Flag: BOOLEAN;
     Frag~id: DESIGNATOR;
     Local~addr: {VECTOR~OF Byte a | LENGTH(a) = Addr~size};
     Inet~addr: STRUCT~OF(Net~id net; Local~addr la);
     Data: VECTOR~OF Byte;
     Type~of~serv: STRUCT~OF(CHAR precedence; Flag strm;
                     CHAR reliability; Flag svsr, speed);
     Packet: STRUCT~OF(Quartet v, ihl; Type~of~serv tos;
             Half~wd totln, indent; Flag df, mf; Offset fo;
             Byte ttl, pcol; Inet~addr sa, da; Data opt, d);

PARAMETERS

     Quartet Version;
     INTEGER Max~offset;
     SET~OF Byte Known~pcols;
     SET~OF Byte Known~nets;

DEFINITIONS

     BOOLEAN Invalid~id(Frag~id fd) IS NOT(fd INSET Current~ids());

EXTERNALREFS

     FROM Phys~net:
        OVFUN Remove~fm~net(Inet~addr da) -> Packet p;
```

```
      INTEGER Byte~size;                $(Currently 8)
      INTEGER Addr~size;                $(Currently 3)
      INTEGER Word~size;                $(Currently 4)

   FROM Inet~rt:
      INTEGER Min~inet~hdr;          $(Currently 20 bytes)


FUNCTIONS

VFUN Offsets(Half~wd id; Byte pcol; Inet~addr sa, da) ->
                                                  SET~OF Offset o;
   $(The description calls for the most recent arrival to be
     used so there is no need to store duplicate information
     if two fragments arrive with the same offset.)
   HIDDEN;
   DERIVATION o = {};

VFUN In~frags(Frag~id fd) -> Packet p;
   HIDDEN;
   INITIALLY p = ?;

VFUN Current~ids() -> SET~OF Frag~id id;
   HIDDEN;
   DERIVATION id = { };

OFUN Recv~pkt(Inet~addr da);
   EXCEPTIONS
      RESOURCE~ERROR;

   EFFECTS
      fd = NEW(Frag~id);
      'Current~ids() = {fd} UNION Current~ids();
      'In~frags(fd) = EFFECTS~OF Remove~fr~net(da);

OFUN Set~offset(Frag~id fd);
   EXCEPTIONS
      Invalid~id(fd);

   EFFECTS
      Offsets(In~frags(fd).id, In~frags(fd).pcol, In~frags(fd).sa,
              In~frags(fd).da) = In~frags(fd).fo;

OFUN Destroy(Frag~id fd);
   EXCEPTIONS
      Invalid~id(fd);

   EFFECTS
      'Current~ids() = Current~ids() DIFF {fd};

END~MODULE
```

7. IMPLEMENTATION

The specifications given above do not define the implementation details to get a runable version of the IP. In fact an attempt was made while writing the specifications to leave individual implementors as much freedom as possible. The final stages of HDM are the implementation stages and each specific implementation would need to carry them out.

In writing the specifications an attempt has been made to make the difference between a host and a gateway, at the internet level, as transparent as possible. For a host all of the modules defined will be needed to some extent; although a particular implementation may make some simplifying assumptions (e.g. all protocols above their IP implementation will pass in packets which do not need to be fragmented to be sent out using the local network). A gateway can be implemented using only a few of the modules: the Send~frag, Recv~frag and Inet~rt modules. Since gateways do not do reassembly they will not need Reassemble. The Gateway-Gateway protocol would need to be refined since the specifications give only an interface to it.

Implementors do not have complete independence from one another. Many variables are exported from one module to another. The methodology also requires certain cross-module checking to be done to show that the implementation maintains the specified properties. The methodology also assumes that a common mechanism would exist for raising and handling exceptions. There also needs to be an agreed upon representation for built-in types (e.g. ASCII vs. EBCDIC). Most of these restrictions are

the type that a common compiler would introduce. Since this cannot be assumed in an internet environment the specification techniques should be enhanced to allow such details to be specified.

The major problem which will occur in implementation is in areas where the specification cannot be implemented in a cost-effective way. The internet header checksum presents this type of problem. Ideally the checksum would be a flag which is either true or false. As we have neither perfect hardware nor perfect transmission media, this cannot be implemented, only approximated. As specification techniques improve it should be possible to include in the requirements the level of confidence desired and include it as a part of the specification.

The normal assumption made in a computer system is that the value read from a memory location is the same as the previous value stored there. The hardware is not perfect the system incorporates error checking/correcting circuits to bring the system reliability to the point where users normally trust the data from memory. A tradeoff is made between cost and reliability. Similar tradeoffs must be made when implementing protocols. In a manner analogous to parity the IP concludes that a datagram is "good" if it passes the checksum test. The degree of assurance is an implementation decision question. The IP description recognizes this and provides the following guidance:

> The checksum field is the 16 bit one's complement of the one's complement of all the 16 bit words in the header... This is a simple to compute checksum and experimental evidence indicates it is adequate, but it is provisional and may be replaced by a CRC procedure, depending on further evidence [Pos80].

Attempting to include the checksum as a function of the representation of the fields in the header runs into a problem with the specification technique. HDM defines the world in terms of strongly typed objects. Since the actual bit representation is not available the checksum algorithm cannot be defined except at a very low level (a level at which the total datagram is represented as a collection of bits). This is consistent since the checksum algorithm does violate the concept of strong typing. (HDM is not alone in having the particular problem; most specification techniques depend on strong typing.)

The problem of checksumming unfortunately keeps cascading upward. The Transmission Control Protocol (TCP), which runs on top of the IP, also checksums. Because of this it cannot use the interface to the IP which provides access only to virtual objects but must have access to the actual physical representations.

The UDP will have a similar problem with the user interface to the IP specified in section 4 even though that interface is intended to provide true datagram service. The UDP provides a checksum; therefore, it too must have access to the physical representations of the objects.

8. PROBLEMS ENCOUNTERED IN THE SPECIFICATION PROCESS

a. Problems Due to the Protocol Description

1. Physical addresses

Using physical addresses at the IP interfaces presents some problems in the specifications. Since an address is visible to the protocols above IP, any algorithm used to supply a mix of equivalent physical addresses would have to be specified as a part of the visible interface of IP because the effects of this algorithm could be used by the protocol above IP. (It may not be a recommended practice to use knowledge of this algorithm but systems programmers are well-known for taking advantage of everything they can if they feel they can use it to increase "efficiency". Since the algorithm could be so used, the specification, to correctly model the IP, would have to reflect the algorithm's visibility.) Any change of this algorithm would require reverification of all levels above to which this is visible. The current specification of IP is such that SEND can supply an address (which must be validated) but if it does not supply a source address a default will be used at the highest level. Those higher level protocols which need to use physical addresses will use the Phys addr level as the user interface.

2. Problems with Return Route Option (RRO)

RRO causes a problem in decomposing the system into levels due to the possibility of an undescribed interaction between RRO and the DF flag. If a packet arrives at a gateway with RRO requested, the DF flag

set and is at the maximum packet size for the next net it is to enter the IP description does not give a well defined action to be taken. Two actions are possible: discard the packet (i.e. add the RRO then, upon discovering the packet cannot be sent on without fragmentation, discard it as called out in the description) or send it on without adding the route information. The specifications take the first action.

RRO can be of an unbounded length but the header can only be of a finite length. The description did not state what happened in the event of overflow. It is also possible for the return route to have "holes" in it. It could overflow at one point but have room at a later point in its route because the SRO can shrink. In both cases information is lost and it is not possible to differentiate between a datagram with a complete route and one with missing addresses. A reasonable action seems to be to discard overflow and generate a GER. This is the approach specified but it does not completely resolve the difficulties. Since datagram transmission is unreliable the GER may be lost so a host receiving a datagram with the RRO cannot be sure that some addresses have not been lost on the way.

A possible change would be to generate a separate datagram (similar to a General Error Report (GER)) to send the return route. Some of these may be lost as the datagram protocol is not a reliable transmission protocol but this system does not have a maximum number of addresses it can accomodate in the return route, whereas the current system can handle a maximum of nine addresses. It also solves all of the problems detailed above with header overflow. Although some of the

route may be lost, it should be possible to adequately reconstruct from the pieces that arrive if the error free thruput of the system is reasonable. The current system is not reliable due to overflow problems and overflow, once it occurs, will continue.

3. The visibility of fragmentation

• There are four fields in the IP header dealing with fragmentation (Indentification (id), More fragments flag, Don't fragment (DF) flag and the fragment offset). In some sense fragmentation should not appear at the user interface to IP. It is not part of the host-host addressing but a gateway-host interaction; it should be hidden by the IP interface. Unfortunately, the id field and the DF flag need to be available at the user interface as the protocol currently stands.

At first it seemed that the DF flag could be eliminated by keeping knowledge within the IP as to whether or not a host has reassembly capability. However, there is at least one case where this status can change. The net could be reloading a system which had crashed and its bootstrap loader may not be able to handle fragmentation although the normal system could [Pos80c].

The id field is an optional field at the user interface; if not supplied the IP must generate one. One clarification that should be made is that a particular protocol must either always supply the id field or else never supply it. The above specifications have this restriction though not having it is even easier to specify. It is added

to make unnecessary much complication which would be required in an implementation in which ids could be supplied or not supplied by the higher level protocol at random.

If a higher level protocol is suppling ids to the IP care must be exercised to prevent reassembly problems. Since the concatenation of source address|destination_address|protocol|id must uniquely identify a packet for purposes of fragment reassembly, it requires that multiple instantiations of the same higher level protocol which supplies the id must cooperate to prevent a possible fragment mix-up.

b. Problems Due to the Methodology

1. Data representation

For the most part the data structures provided by Special proved adequate to the task. The major problem occurred in representing the network as it appears to the IP. The existence of multisets as a predefined type would have made the specification simpler and easier to understand.

2. Assertion language

A more serious problem is the lack of a unified approach to handle the global assertion about the system. No tools exist to handle such assertions except for those that check the assertion for the KSOS security model. This shortcoming is well recognized by SRI and work is being done to give HDM the capability to handle the proof steps required by the v0 stage of the methodology.

It would also be useful to organize the assertions so that properties about the system could be easily found. The specification should state the axioms which are the basis of the proof and the assertions which the system being developed is to maintain.

3. Specification environment

Another problem which exists with using HDM is a lack of unity in its support tools. The difficulties experienced in maintaining consistancy among modules and versions reaffirmed the author's belief that an integrated environment is needed to develop software. SRI is working to produce an integrated specification and verification environment but the results of the current SRI effort were not available.

4. Exception handling

The HDM model for exception handling is that the only observable effect when a function raises an exception is the passage of time. In the protocol area this may need to be modified. Example - packet gets duplicated, one gets through, other gets an exception.

9. CONCLUSIONS

Formal specification techniques can be used to specify communications protocols although there are many areas in which the specification techniques must be improved to allow complete specification and verification.

HDM provides a good methodological framework in which to investigate protocol specification. Many shortcomings exist in the machine support for HDM but the whole area of verification environments is still a field of research. Planning is underway to start an effort to develop a reasonable environment based on the concepts of a number of verification systems each of which has its strong and weak points.

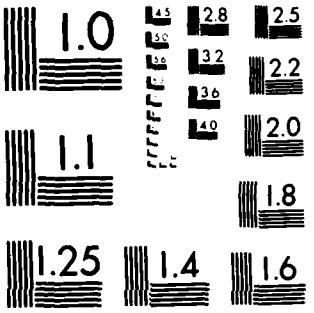
As noted above, HDM also has weaknesses as a methodology for developing protocols. Work is needed to either integrate parts of other existing efforts or develop improved techniques where no existing methodology solves the problem.

There are three clear areas in which progress is needed to allow more complete formal specifications of protocols. First, mathematical modeling techniques for protocols must be improved so that the requirements can be completely and clearly expressed. Second, specification techniques must be augmented to include time as a parameter. The work on temporal logic being done at SRI and Stanford University [Lam80, HO80] is a start in this area. Third, specifications must be able to handle details such as checksums (which "pull apart" data types) in a reasonable way.

The specification of a protocol is an evolving process for two reasons: improvements in specification techniques and changes in protocol functionality. Both of these require modifications to the protocol's formal specification. One definite requirement will be a good programming language to keep the amount of work tractable.

AD-A100 189 ON THE FORMAL SPECIFICATION OF COMPUTER COMMUNICATION PROTOCOLS(U) MARYLAND UNIV COLLEGE PARK DEPT OF COMPUTER SCIENCE K F SHOTTING DEC 80 TR-973 2/2

UNCLASSIFIED AFOSR-TR-81-0491 AFOSR-78-3654 F/G 9/2 NL



Finally it would be useful to implement a version of the IP according to the formal specification then verify the implementation. It would be interesting to compare the results of such an effort with other IP implementations. Major areas for comparison could include time to implement, time to test, performance characteristics, and problems found in the running code -- both in functionality covered by the formal specifications and also in areas not so covered.

GLOSSARY

DF	Don't Fragment flag.
FTP	File Transfer Protocol
HDM	Hierarchical Development Methodology.
IHL	Internet Header Length - length of the internet protocol header in octets.
IP	(DoD Standard) Internet Protocol.
KSOS	Kernelized Secure Operating System.
MF	More-Fragments flag.
MTU	Maximum transmission Unit - maximum size of a fragment which a network can transport.
NCP	Network Control Program
PSOS	Provably Secure Operating System.
PUP	An Internetwork Architecture of the Xerox Palo Alto Research Center.
RRO	Return Route Option - provide a record of the route this datagram took in transmission. Also interpretated as Record Route Option.
SPECIAL	SPECIfication and Assertion Language - A nonprocedural language associated with HDM.
SRO	Source Route Option - provide a list of addresses a datagram is to be routed through.
TCP	(DoD Standard) Transmission Control Protocol.
TOS	Type Of Service - Field in the IP header.
TTL	Time To Live - Field in IP header.
UDP	User Datagram Protocol.

BIBLIOGRAPHY

Bog80

Boggs, D., et. al., "Pup: An Internetwork Architecture", IEEE Transactions on Communications, pp. 612-624, Apr 1980.

BD78

Berson, T. and Drongowski, P., "Formal Specifications for 6.0 UNIX", Version 3.0, WDL-TR7822, Jun 1978.

BM78

Boyer, R. S., and Moore, J. S., "A Formal Semantics for the SRI Program Design Methodology", Technical Report, SRI Computer Science Laboratory, Nov 1978.

Cer78

Cerf, V., "IEN 48- The Catenet Model for Internetworking", Jul 1978.

Coh79

Cohen, D., ed., "IEN 126- Summary of the ARPAnet/Ethernet Community Meeting", Nov 1979.

Coh80

Cohen, D., "IEN 137- On Holy Wars and a Plea for Peace", Apr 1980.

CGHM80

Cheheyl, M., Gasser, M., Huff, G. and Millen, J., "Secure System Specification and Verification: Survey of Methodologies", MTR-3904, The MITRE Corp., Bedford, MA., Feb 1980.

CK78

Cerf, V. G. and Kirstein, P. T., "Issues in Packet-Network Interconnection", Proceedings of the IEEE, Nov 1978.

Dij68

Dijkstra, E., "The Structure of THE Multiprogramming System", CACM 11, pp. 341-346, May 1968.

Div80

Divito, B., Private communication.

FAC78

"KSOS Computer Program Development Specification (Type B5): Security Kernel", WDL-TR7932, Ford Aerospace and Communications Corp., Palo Alto, CA, Sept. 1978.

Goo77
Good, D. I., et. al., "Constructing Verifiably Reliable and Secure Communications Processing Systems", ICSCA-CMP-6, The University of Texas at Austin, Jan. 1977.

Goo78
Good, D. I., et. al., "Gypsy 2.0", ICSCA-CMP-10, The University of Texas at Austin, Jul. 1978.

GW79
Gerhart, S., and Wile, D., "The DELTA Experiment: Specification and Verification of a Multiple-User File Updating Module", Proceedings of the Specifications of Reliable Software Conference, IEEE Computer Society, Apr. 1979.

HO80
Hailpern, B. and Owicki, S., "Verifying Network Protocols Using Temporal Logic", Computer Systems Laboratory, Stanford University, May 1980.

Kem79
Kemmerer, R., "Verification of the UCLA Security Kernel: Abstract Model, Mappings, Theorem Generation and Proof", PhD. Thesis, UCLA, 1979.

Lam80
Lamport, L., "'Sometimes' is sometimes 'not never': On the Temporal Logic of Programs", 7th ACM Symposium on the Principles of Programming Languages, pp. 174-185, Jan 1980.

Luc79
Luckham, D. C., et. al., "Stanford Pascal Verifier Users Manual", STAN-CS-79-731, Computer Science Dept., Stanford University, Mar 1979.

LRS79
Levitt, K., Robinson, L., and Silverberg, B., "The HDM Handbook, Volume III: A Detailed Example of the Use of HDM", SRI Report on Contract N00123-76-C-0195, Jun 1979.

Neu76
Neumann, P. G., Private communication.

Neu77
Neumann, P. G., et. al., "A Provably Secure Operating: The System, Its Applications, and Proofs", Final Report - SRI Project 2581, Menlo Park, Ca., Feb. 1977.

Par72
Parnas, D. L., "A Technique for Software Module Specification with Examples", CACM 15, pp. 330-336, May 1972.

Pol80
Polak, W., "Theory of Compiler Specification and Verification", PhD Thesis, Stanford University, Apr 1980.

Pos79
Postel, J., "IEN 88- User Datagram Protocol", May 1979.

Pos80a
Postel, J., ed., "IEN 128- DoD Standard Internet Protocol", Jan 1980.

Pos80b
Postel, J., ed., "IEN 129- DoD Standard Transmission Control Protocol", Jan 1980.

Pos80c
Postel, J., Private communication.

Pos80d
Postel, J., "Internetwork Protocol Approaches", IEEE Transactions on Communications, pp. 612-624, Apr 1980.

Rob79
Robinson, L., "The HDM Handbook, Volume I: The Foundations of HDM", SRI Report on Contract N00123-76-C-0195, Jun 1979.

RR77
Roubine, O., and Robinson, L., Special Reference Manual, Stanford Research Institute Technical Report, CSC 45, Jan. 1977.

Str79
Strazisar, V. "IEN 109- How to Build a Gateway", Aug. 1979.

Sun78
Sunshine, C., "Survey of Protocol Definition and Verification Techniques", ACM Computer Communication Review, Jul 1978.

Sun79
Sunshine, C., "Formal Techniques for Protocol Specification and Verification", IEEE Computer Society Magazine, Sep 1979.

Sun80
Sunshine, C., "Axioms for the Alternating Bit Protocol", Affirm Memo-17-Cas, Feb 1980.

SLR79
Silverberg, B., Levitt, K. and Robinson, L., "The HDM Handbook, Volume II: The Languages and Tools of HDM", SRI Report on Contract N00123-76-C-0195, Jun 1979.

Tho79
Thompson, D., ed., "AFFIRM Reference Manual", USC Information Sciences Institute, Nov 1979.

TAL79
Thareja, A., Agrawala, A., and Larsen, R., "Formal Approaches to Protocol design and Implementation: A Survey", TR-840, University of Maryland, Dec 1979.

Wel76
Wells, P., "Specification and Implementation of a Verifiable Communications System", ICSCA-CMP-4, The University of Texas at Austin, Dec 1976.

Wen76
Wensley, J., Green, M., Levitt, K., Shostak, R., "The Design, Analysis, and Verification of the SIFT Fault-Tolerant System", Second International Conference on Software Engineering, San Francisco, CA, Oct 1976.

X.75
CCITT Recommendation X.75 (provisionally adopted), "X.75 Terminal and transit call control procedures and data transfer system on international circuits between packet-switched data networks", International Telecommunication Union, Geneva, 1979.

END

DATE
FILMED

5 -83

DTIC